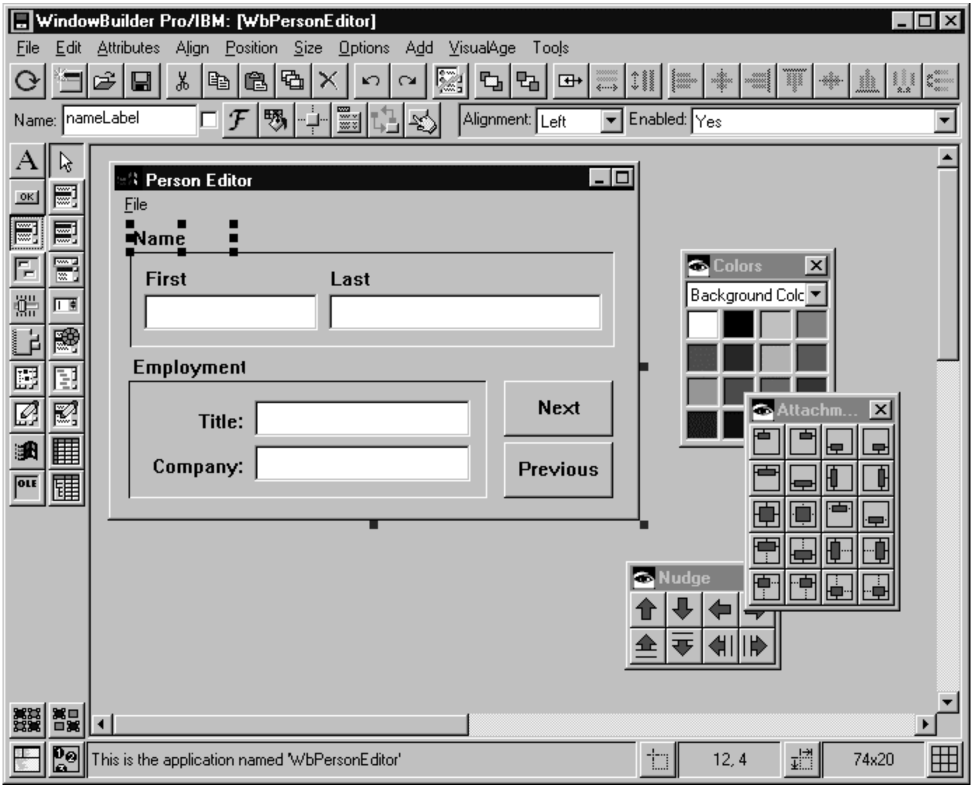
WINDOWBUILDER® PRO

Tutorial and Reference Guide



WindowBuilder Pro/IBM: [WbPersonEditor]

Name: nameLabel

Alignment: Left

Enabled: Yes

This is the application named 'WbPersonEditor'

12, 4

74x20

INSTANTIATIONS, INC
SMALLTALK SYSTEMS DIVISION

Instantiations License Agreement

Second Edition July 1998. Printed in Oregon.
Instantiations, Inc., Smalltalk Systems Division
16004 SW Tualatin-Sherwood Road, Suite 435
Sherwood, OR 97140-8378

Contents

Acknowledgments

Software Design & Development:

Eric Clayberg

Manual

Eric Clayberg

Testing

William Dargel, Kalpana Krishnaswami, Jeff Odell, Steve Parker, Tom Petersen, Dan Rubel, Gordon Sheppard, Enoch Sower, S. Sridhar, Solveig Viste, Chris Wolcott

We would like to thank the following people. Without their hard work, help, advice, support and debugging skills, this product would never have seen the light of day. Thanks!

Joseph Acero, Bill Baer, Robert Benson, Carter Blitch, Rob Brown, Pat Caudill, James Chan, Eric Clayberg, Karen Clayberg, Steven Daniels, William Dargel, Scott Day, Stef van Dijk, Shawn Elliot, Juanita Ewing, Marten Feldtmann, Dina Fischer, Max-Pieter Fränkel, Amarjeet Garewal, Suman Goel, John Hansen, Steve Harris, Chris Hayes, Scott Herndon, Bill Hertha, Hal Hildebrand, Pat Huff, Dan Kehn, Ed Klimas, Alan Knight, Kalpana Krishnaswami, Chamond Liu, Byron Long, Jasmin McCabe, Cynthia McCrickard, Paul McDonough, Steve Messick, Dave Mitchell, Carmelo Montalbano, Martin Nally, Jimmy Nguyen, Jeff Odell, Steve Parker, Joseph Pelrine, Tom Petersen, Hudson Philips, Rene' Plourde, Dan Rubel, Kristi Rudolph, Rick Runyan, Dan Shafer, Gordon Sheppard, Ed Shirk, Sames Shuster, Mike Silverstein, Enoch Sower, S. Sridhar, Mike Taylor, Ken Thompson, Steve Robinson, Rick Runyan, David N. Smith, Solveig Viste, David Whiteman, Marlin Wilson, Allen Wirfs-Brock, Chris Wolcott, Robert Yerex, Sherwood Zern

Chapter 1 Introduction

Welcome to WindowBuilder Pro! You have purchased the most advanced graphical user interface (GUI) builder available. Read on to learn about what WindowBuilder Pro does, and how it can dramatically increase your VisualAge development productivity.

What is WindowBuilder Pro?

WindowBuilder Pro is a complete GUI builder. All the tools that you need to create user interfaces are contained in WindowBuilder Pro. Just draw the windows with a mouse as you would with a paint program. Add buttons, list boxes, and scroll bars. Arrange, resize, and rearrange these screen elements until you are satisfied. When you get the screen looking the way you want, WindowBuilder Pro generates the necessary Smalltalk code for you. Add the rest of your application code, and your program is complete. You never have to write any user interface (UI) code. If you later find that you need to make changes to the UI, you can make the changes right on the screen, and WindowBuilder Pro will recreate the code automatically. WindowBuilder Pro takes care of all the down-and-dirty work of writing UI code, and allows you to concentrate on writing the application code. You should find that your program development time noticeably decreases.

What you should already know

To be successful using WindowBuilder Pro, you need

- a working knowledge of VisualAge Smalltalk
- familiarity with components and navigation of the operating system you are using

This manual assumes that you have a functional knowledge of VisualAge. WindowBuilder Pro allows you to generate user interfaces without knowing how to program in Smalltalk. However, you will not be able to write the code necessary to create a fully functioning application. If you are new to Smalltalk, you should work through the examples provided in the *IBM Smalltalk Programmer’s Reference* before proceeding with WindowBuilder Pro.

You need to be familiar with operating system interface components such as dialog boxes, buttons, and menus. You should also understand the mouse concepts of pointing and clicking, as well as the text manipulation commands for your system.

History of WindowBuilder Pro

The first version of WindowBuilder was called Widgets/V, and was introduced in 1990 by Cooper & Peters. It ran on the Macintosh and the IBM PC under DOS. In 1991 Cooper & Peters built a version for Microsoft Windows and renamed the product WindowBuilder. Objectshare Systems, took over the development of WindowBuilder in 1992. WindowBuilder Pro was developed by Objectshare Systems and appeared first for Digitalk Smalltalk in 1993.

WindowBuilder Pro for VisualAge first appeared in 1994 and the briefly became a product of ParcPlace-Digitalk following its acquisition of Objectshare in 1996. In 1997 Instantiations' Smalltalk Systems Division acquired rights to WindowBuilder Pro for VisualAge and now develops, markets and supports it. It is notable that the original development team of WindowBuilder Pro for VisualAge has followed the product to each of its new homes and continues to work on it today!

What's New?

There are dozens of new features that have been added to the product (many at the suggestion of users like yourself) in recent releases. In no particular order they are:

- Support for new IBM Smalltalk widgets
  - EwProgressBar
  - EwToolBar
  - CwSash (splitbar)
- New WbEnhancedText widget
  - Character and field-level validation
  - Password style
  - Left, right and center justification
- New WbObjectComboBox widget
  - Object-oriented version of CwComboBox
  - Works with any arbitrary objects, not just strings
  - #printSelector attribute specifies how the obects will be displayed
  - Supports type ahead object matching in text edit mode

- Support for Windows 95 widgets
    - CwStatusBar
    - CwToolBar
    - CwTabStrip
    - CwTreeView
    - CwProgressBar
    - CwTrackBar
- Support for OLE/ActiveX
    - OleClient
    - OleControl
    - Wrappered OLE/ActiveX widgets (AbtOleExtendedWidget subclasses)
- Runtime Unix Support
    - Develop under Windows or OS/2
    - Deploy under Windows, OS/2 or any VA Unix platform
- Enhanced Integration with VisualAge
    - WBPro windows may be embedded within VisualAge windows as visual components
    - Conceptually similar to nested applications within WBPro or CompositePanes in VSE
    - WBPro is now the ideal environment for creating complex, reusable visual parts for VisualAge
- Lightweight visual programming (e.g., configure callbacks and event handlers via drag drop like the VisualAge Composition Editor)
    - Popup connect menu listing callbacks and events
    - Drag connect the source to the target
    - Popup message menu on target
    - Resultant callback/event handlers can be viewed with Callback Editor
- New Callback Editor
    - Widgets are displayed as graphical tree
    - Widgets may be displayed hierarchically, alphabetically (by name or type)
    - Widgets may be filtered by type (e.g., view just the CwPushButtons)
    - Multiple widget select (create callbacks/event handlers on multiple widgets simultaneously)
    - Multiple handler select (change receiver/selector/clientdata simultaneously)
    - Handlers may be ordered via up/down buttons
    - Handlers may be zero (unary) or one-argument methods in addition to the standard three-argument methods. For one-argument callbacks, the default argument that is passed is the originating widget. Specifying a unary selector as the client data will cause the attribute of the originating widget specified by that selector to be passed as the argument (e.g., specifying `#selectedItem` as the client data for the Single Selection Callback of a listbox will cause the selected item to be passed as the argument).
    - Built-in event handler editing via embedded code browser

- New Help Editor
    - Specify tooltips (mini / hover help) for any widget
    - Specify platform help files and help topic IDs for any widget
    - New WbPlatformHelpExample provided
    - Mini / hover help enhanced to work with EwToolBar tools
- Drag drop tab order setting
    - In “Show Tab/Z-Order” mode, the tags are live and may be dragged from one widget to another
- Graphically enhanced tab/z-order editor, call-out editor, and drag-drop editor
    - Widgets are displayed graphically in list (e.g., icon and name)
    - Status is displayed graphically
- New toolbar buttons: Morph, Undo, Redo, Select All, etc.
- New menu layout (e.g., Align, Position and Sizing functions have been separated)
- Context sensitive popup menus everywhere (in layout area and on numerous toolbar buttons)
    - Popup widget menu reflects the type and number of selected widgets
    - Popup morph menu lists morphing types for the selected widget(s)
    - Popup undo and redo menu list undoable and redoable actions
    - Popup select all menu lists are widget types in layout (e.g., makes it easy to select all CwLabels)
    - Popup open menu lists recently accessed classes
- Dynamic, context sensitive style selection
    - Style comboboxes now have identifying labels
    - Right-clicking on style comboboxes allows you to change the displayed style choices
- New Generic Attribute Editor, Template Editor and Property Editor
    - Graphically enhanced (e.g., widgets are displayed with icons and labels)
    - Table widget is now used for all attribute/property setting
- Enhanced Attachment Editor
    - More default styles
    - New thumbnail before and after views
- Enhanced Color Editor
    - New palette of the 16 “primary” colors
- New floating tool windows
    - Color tool (this is a mini version of the Color Editor)
    - Attachment tool (this is a mini version of the Attachment Editor)
    - Nudge (move/size by pixel) tool
    - Tab & Z-Order tool (bring to front, send to back, etc.)
    - Widget Selection tool (select multiple widgets at any level of the widget hierarchy)
    - They remember their last size and position each time they are opened

- New Layout features
  - New Drag & Drop Reparenting option allows widgets to be reparented simply by dragging them from one parent to another. For example, a widget may be dragged from the top level form into a nested form without the need to cut and paste it. Likewise, a table widget may be dragged into a scrolled window to make it scrollable.
  - Side handles are now available for selected widgets (in addition to the existing corner handles). This gives more precise control over resizing a widget in only one direction
  - New Vertical and Horizontal Packing functions make it easy to cluster groups of widget together
  - If the ALT key is held down while performing a horizontal or vertical widget alignment, only the specified sides of the widgets will be aligned while the opposite sides will not move. This will cause the widgets to grow or shrink in size (as opposed to moving and retaining their original sizes)
- New Timer support protocols in WbApplication
  - Easily set up (and remove) timers
  - New #timer event
  - Use #startTimer:period: to create a timer
  - Use #stopTimer: to stop a timer
- Support for Icons as graphical labels
  - New smart WbIcon subclass of CgIcon
  - Load icons from .ICO files or from resource DLLs
  - Pixmap Editor is now the Graphics Editor and can be used to select Pixmaps or Icons
- New WbLabeledImage runtime support class
  - Combines an image and a label into a single renderable object
  - Supports EwRenderContext interface
  - Horizontal or vertical orientations supported
  - Create toolbars with labeled buttons - see the WbLabeledImageExample class for an example
  - Create fancy iconic lists and tables
- More Code Generation Options
  - Optionally use generic IBM Smalltalk code generation
  - Optionally generate EtWindow subclasses
- Runtime IC Generation
  - New "Runtime ICs" submenu is available from the Transcript's WindowBuilder menu
  - Generate ICs for all WindowBuilder & WidgetKit components
  - Generate ICs for either development or runtime images
- Minor window enhancements
  - Any window can be made to float above the main WindowBuilder Pro window

- All windows have their own icons (makes it easy to distinguish between them in the task bar)
- Splitbars are used where appropriate

How this manual is organized

Chapter 2 covers installation of the product.

Chapter 3 of this manual describes the process of creating user interfaces with WindowBuilder Pro. In it you put together a small “Hello World” program, and become familiar with the WindowBuilder Pro environment. You learn the basics of writing code that enables a WindowBuilder Pro interface to interact with other Smalltalk objects.

Chapters 4 and 5 are a functional reference to the WindowBuilder interface.

Chapter 6 describes the code writing process in detail. In it you learn about writing different kinds of callbacks, passing values to and from the UI, and about the widget hierarchy.

In Chapter 7, you construct a simple application, using what you have learned in the first four chapters of this manual.

Chapter 8 is a comprehensive reference to all of WindowBuilder Pro’s commands.

Chapter 9 is an overview of the Common Widgets subsystem in VisualAge.

Chapter 10 is a detailed discussion of callbacks and event handlers.

Chapter 11 is a detailed discussion of each of the standard widgets in VisualAge.

Chapter 12 is a complete widget encyclopedia detailing the protocols and events that each widget responds to.

Chapter 13 is a reference to the WbApplication window framework provided with WindowBuilder Pro.

Appendix A is a guide to customizing WindowBuilder Pro.

Appendix B discusses the extended widget framework in VisualAge

Appendix C describes the user interface process model used by VisualAge.

Appendix D details behavioral differences between each platform supported by VisualAge.

Appendix E discusses the WindowBuilder Pro’s integration with VisualAge.

Typographic Conventions

This manual uses the following typographic conventions.

Example of convention	Used for
`printString`	Smalltalk code
Style	words that you type in
C:\VAST\ABT.EXE	file names
input name	placeholders for your input
CNTRL+S	key combinations

The symbol is used to indicate that a feature is Windows 95/98 or Windows NT 4.0/5.0 specific. An example would be OLE/ActiveX support.

Technical Support

We provide 30 days of free support for WindowBuilder Pro to *registered* users, Monday-Friday, from 8:30 AM to 2:30 PM, Pacific Standard Time. If you have purchased WindowBuilder Pro but have not registered it, we can not provide support. Additional support contracts may be purchased for a nominal fee and include maintenance upgrades and bug fixes - contact Instantiations' Smalltalk Systems Division for details. We prefer to handle support questions via e-mail at **support@smalltalksystems.com** or via our Internet newsgroup at **news://nt1.netsmart.com/sts.smalltalk**. You can also call in to our order and support system at **800-808-3737**. When contacting us electronically or when calling in a support request please provide as much information as possible. The following details will be needed for us to provide prompt support:

1. Version and serial number of your WindowBuilder Pro product
2. Version number of the VisualAge product you are using
3. Version number of the operating system you are using
4. Any special information about your configuration.

We anticipate some confusion over the ownership of problems associated with WindowBuilder Pro. Please keep in mind that the widgets provided in VisualAge are products of IBM, not Instantiations. If your problem is related to widget or base system functionality, please refer to IBM for support. Our technical support is strictly limited to problems in the runtime or development portions of WindowBuilder Pro. Problems with runtime specific behavior of base image widgets should be referred to IBM.

Chapter 2 Installation

This chapter will tell you everything you need to know in order to install WindowBuilder Pro.

Prerequisites

In order to use WindowBuilder Pro, you must be currently a user of VisualAge 4.02 or 4.5. Earlier, beta versions may also work, but are not guaranteed to work.

If you do not have VisualAge on Windows or OS/2 (either standalone or server), you will not be able to use this release of WindowBuilder Pro. Please note that if you are using WindowBuilder Pro with the server version of VisualAge, you must have one license of WindowBuilder Pro for each developer who loads the product from the server.

We assume that you are familiar with the VisualAge development environment and have some experience with programming in Smalltalk.

Installation

Check your disk space. The total required for WindowBuilder Pro installation is 8 MB at minimum. WindowBuilder Pro can be installed either from a CD or from electronic distribution (ZIP) files. The CD includes versions of the product for Windows and OS/2. You will undoubtedly need only a portion of these files for your current environment.

To install WindowBuilder Pro from the CD:

Run SETUP???.EXE off of the CD (where “???” is either “WIN” or “OS2”). You will be asked to enter the path to your VisualAge main directory (e.g., C:\VAST). The WindowBuilder Pro files will be decompressed and installed in the appropriate subdirectories off of the main directory. If you would prefer that the setup program not install into this directory tree, specify an alternative directory as the target. If you specify a directory that does not exist, the setup program will create it for you. The setup program will also automatically create subdirectories under the specified directory that mirror the directories that it would normally install into (make sure that you then move/copy the files to the indicated directories).

To install WindowBuilder Pro from electronic distribution files:

WindowBuilder Pro is distributed electronically as a set of ZIP files. The files should be unzipped and their contents placed into the directories indicated in the next section. The applicable ZIP files are:

WB45VA.ZIP

Contains all of the files needed to install and run WindowBuilder Pro under either Windows or OS/2 (e.g., DAT, CTL and DLLs).

WB45DOC.ZIP

Contains the WB45VA.PDF file. This is the on-line help file in Windows help file format.

File Locations

The following files are installed. Each file should be placed into the appropriate VisualAge subdirectory (indicated for each file). We have indicated whether a file has any operating system dependencies.

READMEWB.RTF (\VAST directory)

Read Me file containing installation instructions and an overview of the product. You should read this file before installing the product.

WB45*VA.DAT (\VAST\IMPORT subdirectory)

This is a VisualAge export library that contains several configuration maps for the various elements of WindowBuilder Pro.

ABT*.CTL (\VAST\FEATURE subdirectory)

These are the control files that add WindowBuilder Pro to the “Load/Unload Features...” list. These are not needed if you use the manual loading option below.

STS40VAW.DLL (\VAST directory - Windows only)

This is a resource-only DLL containing bitmaps and icons used by the Windows version of WindowBuilder Pro. This file should be copied to a directory that is located on the search path described by the PATH variable in AUTOEXEC.BAT (e.g., \VAST).

STS40VAO.DLL (\VAST directory - OS/2 only)

This is a resource-only DLL containing bitmaps and icons used by the OS/2 version of WindowBuilder Pro. This file should be copied to a directory that is located on the search path described by the LIBPATH variable in CONFIG.SYS (e.g., \VAST).

WB45VA.PDF (\VAST directory)

On-Line manual in Adobe Acrobat format. This requires the Adobe Acrobat Reader to view.

Installing WindowBuilder Pro into the VisualAge Image

To install WindowBuilder Pro, bring up the VisualAge image. There are two ways to load WindowBuilder Pro into your image:

Automatic Load

From the Transcript's **Tools** menu, select the **Load/Unload Features**... option. Select "WindowBuilder Pro" from the dialog box. This will load WindowBuilder Pro into your image.

Manual Load

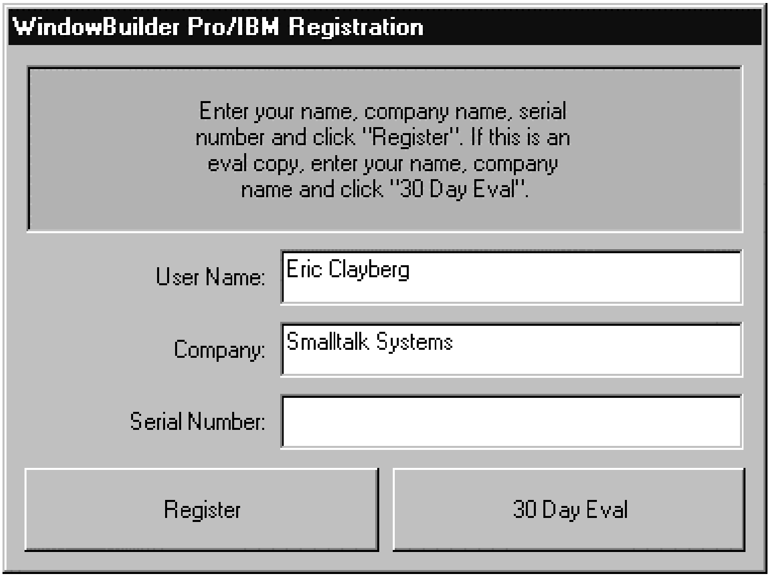
Bring up a Configuration Maps Browser, and import the "WindowBuilder Pro" configuration into the library from the WB45*VA.DAT manager file. Then load this configuration into your image.

Several other configurations may also be loaded:

- "WindowBuilder Pro - Examples" contains all of the WindowBuilder Pro tutorial and example code.
- "WindowBuilder Pro - Tools" adds a tools/debugging menu to WindowBuilder Pro. This would be useful for advanced users

When WindowBuilder Pro has finished loading, a new menu called WindowBuilder will be added to the Transcript menu. You'll launch WindowBuilder Pro and related tools from this menu. WindowBuilder Pro is now loaded, enabled and ready to go!

If this copy of WindowBuilder Pro is a fully licensed copy, you may also register it via the menu item **Register** under the **WindowBuilder** menu. Enter your name, company name, serial number and click the **Register** button.


WindowBuilder Pro/IBM Registration

Enter your name, company name, serial number and click "Register". If this is an eval copy, enter your name, company name and click "30 Day Eval".

User Name: Eric Clayberg

Company: Smalltalk Systems

Serial Number:

If this is an evaluation copy of WindowBuilder Pro and you don't have a valid serial number, enter your name, company name and click the **30 Day Eval** button. You can then use WindowBuilder Pro free for 30 days in order to decide whether you wish to purchase it. You can order the product by calling Instantiations at 800-808-3737.

Chapter 3 Overview

This chapter introduces you to creating window based applications with WindowBuilder Pro. In this chapter you will become familiar with the WindowBuilder Pro environment

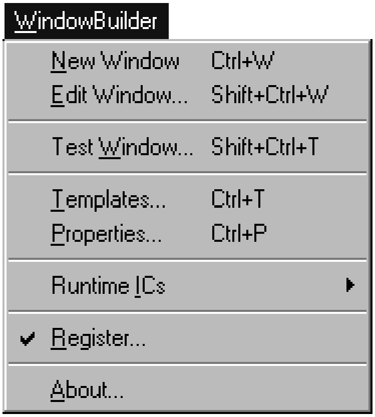
- learn about the process of creating window-based applications
- create a simple application

Starting WindowBuilder Pro

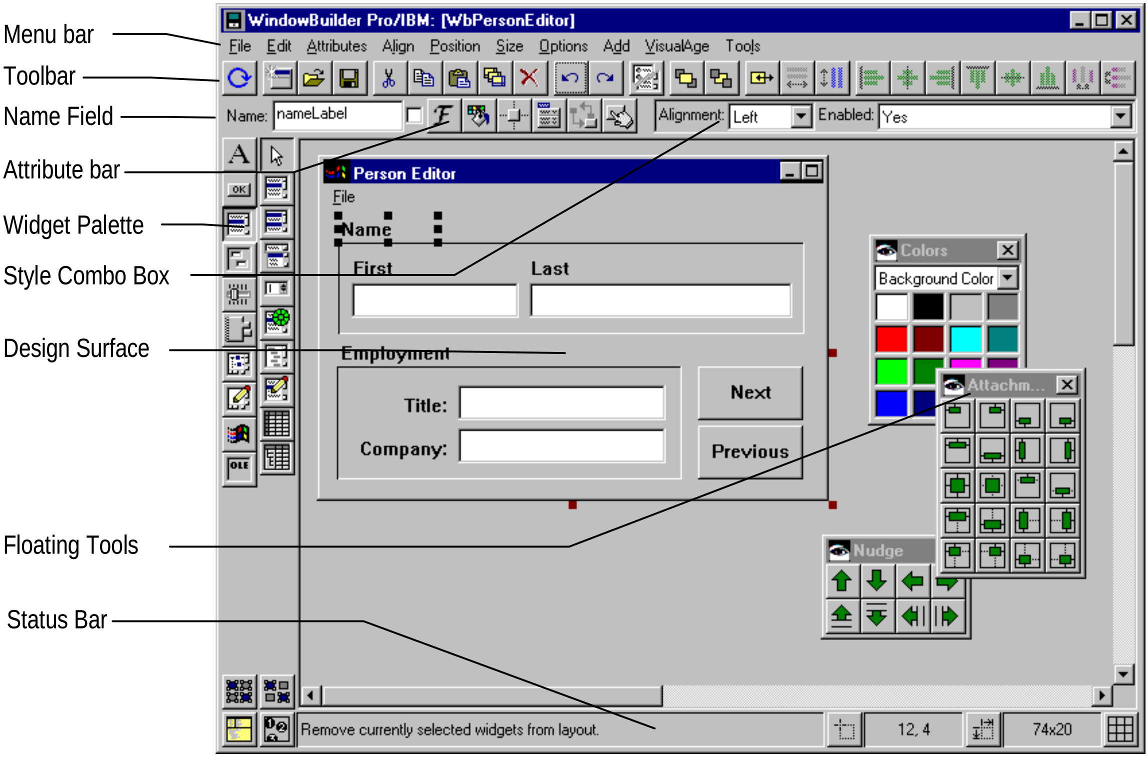
After you have completed installation, start your VisualAge for Smalltalk image. Notice that the installation procedure has added a new menu item, **WindowBuilder**, to the System Transcript menu bar.

To start WindowBuilder Pro:

- Choose **New Window** from the WindowBuilder menu in the System Transcript window.



The WindowBuilder Pro screen appears, as shown on the next page.



Name: nameLabel

Alignment: Left

Enabled: Yes

Background Color

Remove currently selected widgets from layout.

12, 4

74x20

WindowBuilder Pro Main Window

Toolbar Displays icons representing shortcuts for menu commands.

Widget Palette Displays icons representing the widgets available for placement in a window. You can also access these widgets from the Add menu.

Design Surface The work area on which you build your windows.

Menu Bar **G**roups commands and options under text headings.

Attribute Bar Allows you to add behavior to your widgets.

Name Field Allows you to assign a name to a widget.

Style Combo Box Displays style information about a widget.

Floating Tools Provide quick access to color, attachment, tab & z-order, widget selection and fine tuning controls.

Status Bar Provides information on currently selected widgets.

Creating a window-based application

Application development in WindowBuilder Pro comprises two steps:

- Design the user interface
- Attach callback and event handlers to the interface objects

Designing the user interface is the process of placing buttons, menus and other user interface objects (collectively called *widgets*) in a window.

Attaching callbacks and event handlers is the process of associating events with Smalltalk methods. An *event* is a mechanism that notifies the application when the user performs a mouse or keyboard action. An example of an event is clicking the mouse button.

A *callback* is a mechanism that notifies the application that some higher level action is performed on a widget.

An *event handler* is a method that launches when certain operations are performed on window objects. For example, when a button is clicked, an event handler associated with the action “activate” might display a message on the screen.

You learn how these steps actually work in the rest of this section. Read the procedures, then follow along with the example to create a small application consisting of a text widget and a push button. In this application, clicking the push button will cause the message “Hello World” to appear in the text.

Step 1. Design the User Interface

When you start WindowBuilder Pro, a window is displayed on the design surface. You can resize this window by dragging the small red rectangle (called a *handle*), located at the lower right corner of the window. If you make the window larger than the design surface, you can use the scrollbars on the right and bottom of the design surface to scroll across the window.

Placing a Widget in a Window

You add widgets to the window by selecting them from the widget palette, or from the Add menu. Notice that the widget palette has two columns. The left column displays categories of widgets, and the right column displays types of widgets in those categories. You select a widget by first clicking its category icon button in the left column, and then clicking the icon button in the right column that represents the type of widget that you want from that category. For example, to select a check box, first click the button icon in the left column.

OK

When you do so, the right column icons change to display the various types of buttons. Click the check box icon in the right column to select the check box.

☑

Remember that you can also select a widget from the Add menu. You may find it easier to use this method until you have learned to associate the widgets with their icons. Note that if you allow the mouse pointer to linger over any toolbar icon, a popup help window will appear describing what it is.

To place a widget in a window:

1. Select the widget that you want to place in the window by clicking the Category icon in the left column, and the Type icon in the right column.

Or, from the Add menu, choose the menu corresponding to the widget category, then choose the type of widget that you want.

2. Move the pointer to the window. The pointer becomes a cross hair. This indicates that the cursor is loaded with the widget, and ready to place in the window.
3. Position the cross hair where you want the upper left corner of the widget to be. Hold the left mouse button down and drag the cross hair to draw the widget. You can also just click and release the mouse button which will allow WindowBuilder Pro to place the widget on the screen, using the widget's default size.
4. When the widget is the size you want, release the mouse button. The widget appears in the window, surrounded by four small black rectangles (handles).

To resize a widget:

- Point to one of the corner handles, and drag it until the widget is the size you want.
- If you want to resize in a horizontal or vertical direction only, use the sizing handles on the sides of the widget

To move a widget:

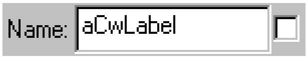
- Point anywhere on the widget other than a handle, and drag the widget to the new location.

Naming and Labeling a Widget

All widgets must have a name. This name is used only as a code reference; it is not visible to the application user. This name is necessary if you want to direct messages to the widget. When a widget is added to the design surface, the system automatically assigns it a default name (generally its type followed by a number). If you want to reference the widget in your code, you will likely want to replace this default name.

To change the name of a widget:

- Type the name in the **Name** field in the Attribute panel.



Name: aCwLabel □

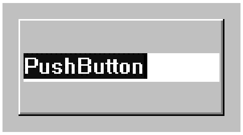
Each widget must have a unique name. If more than one widget shares the same name, unpredictable behavior may result.

A label is the actual text that will display on the face of the widget. Not all widgets can display a label. Widgets that can display a label have, by default, their generic title as their default label. For example, a CwPushButton has “PushButton” as its default label. You can replace the generic title aCwPushButton with your own label. Labeling a widget can be accomplished by directly editing the widget or by using the widget’s attribute editor. There are attribute editors for every kind of widget. There is also a generic attribute editor.

For More Information on Attribute Editors, see Chapter 9, *Command Reference*.

To add a label to a widget by direct editing:

1. Point to the widget, hold the ALT key and click. The widget will then enter direct edit mode.



2. Type the text of the label.

If the widget only supports a single line label (for example, CwPushButton), press ENTER to finish direct editing and see the results. If the widget supports a multi-line label (for example, CwLabel), click anywhere outside of the widget to finish editing (pressing ENTER will start a new line). Pressing ESC will end direct edit mode without accepting any changes. Note that almost all of the widgets in the system support direct editing. For example, a WbScrolledList will allow you to set its contents in direct edit mode. Using direct edit with a CwForm or CwRowColumn widget will allow you to edit its children.

To add a label to a widget by using the widget’s attribute editor:

1. Point to the widget and double-click. The attribute editor for the widget appears.
2. Type the text of the label in the **Label String** field.

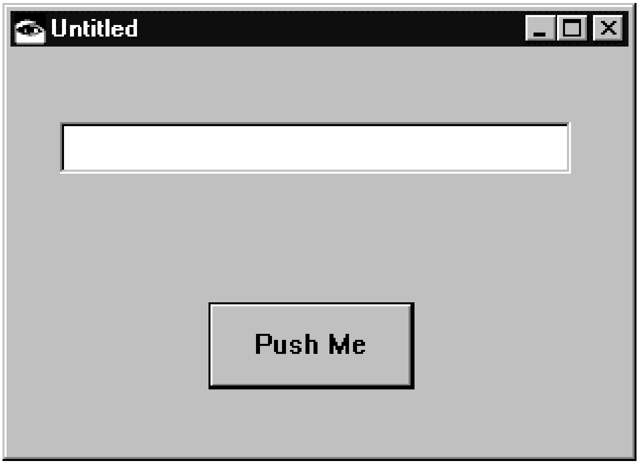
3. Click **Apply** to see the result immediately. When you are satisfied, click **OK** to close the attribute editor.

Designing the “Hello World” Application

Now you have the procedures necessary to draw the widgets for the sample application. You’ll use a CwPushButton and a CwText widget. Place the widgets into a blank window, and add names and labels to them as follows:

Widget Type	**Name**	**Text:**
CwPushButton	pushMeButton	Push Me
CwText	greetingField	(none)

When you are done, the screen should look similar to this:



“Hello World” application

Step 2. Attaching callbacks and event handlers to interface objects

You attach callbacks and event handlers to the currently selected widget by using the Callback Editor.

For example, if you want a message to display in a text widget when the user clicks a button, you select the Activate callback from the Callback List, click **Add**, and type a method name in the Methods: entry field. Standard callback method names in VisualAge are three part selectors of the form “*descriptiveName*:`clientData:callData`:”. You only need to type the descriptive name followed by two colons; WindowBuilder Pro will automatically append “`clientData:callData:`” for you.

In the Class Browser, add the code to display a message in a text widget. The application then knows that it is to display a message in a text widget when the button is clicked. This is how you add behavior to user interface objects.

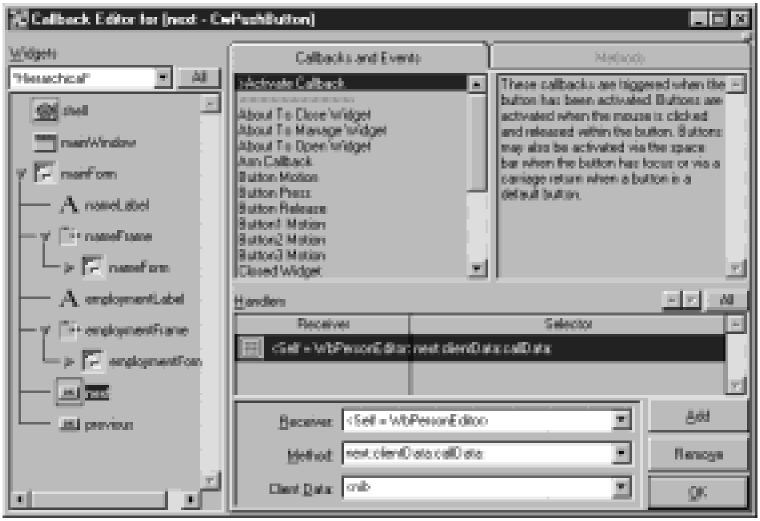
For More Information on the Callback Editor, see Chapter 8, *Command Reference*.

To attach a callback to a widget:

1. Select a widget and click the callback editor toolbar button.

Or, choose **Callbacks**... from the Attribute menu.

The Callback Editor appears. Select the type of callback to which you want the application to respond. A full description of the callback is displayed to the right of the callback list.



Callback Editor for [next - CwPushButton]

These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Handlers

Receiver	Selector
<Self = WbPersonEditor>	next:clientData:callData:

Receiver: <Self = WbPersonEditor>

Method: next:clientData:callData:

Client Data: <nil>

2. Click the **Add** button (or double-click the callback type).
3. Specify the receiver of the callback by selecting a widget from the Receiver list box.

Generally, the receiver is the application itself (“self”). In some cases, you may want the receiver to be some other object, such as one of the other widgets in the window or one of the instance variables of the application (for example, a model object).

4. Type the name of the method in the **Methods**: combo box, followed by two colons. When you type the second colon, the editor automatically fills in the rest of the method name by appending `clientData:callData`: for you.

Zero (unary) and single argument callback handlers are also supported. Leave the second colon off, if you do not want the method name expanded to the standard three argument style.

5. Type client data, if any, in the **Client Data** combo box.

Client data is any arbitrary information that you want to pass to the callback handler (this is the second argument of the callback name that you just defined). Generally, the client data will be nil. Occasionally, you may want to pass a different value such as a String, a Symbol, an Integer, or a Class. It is up to you to decide what to do with this client data in the callback method itself.

Unary callback handlers do not pass along any arguments and thus do not allow client data to be specified.

For one-argument callbacks, the default argument that is passed is the originating widget. Specifying a unary selector as the client data will cause the attribute of the originating widget specified by that selector to be passed as the argument (e.g., specifying `#selectedItem` as the client data for the Single Selection Callback of a listbox will cause the selected item to be passed as the argument).

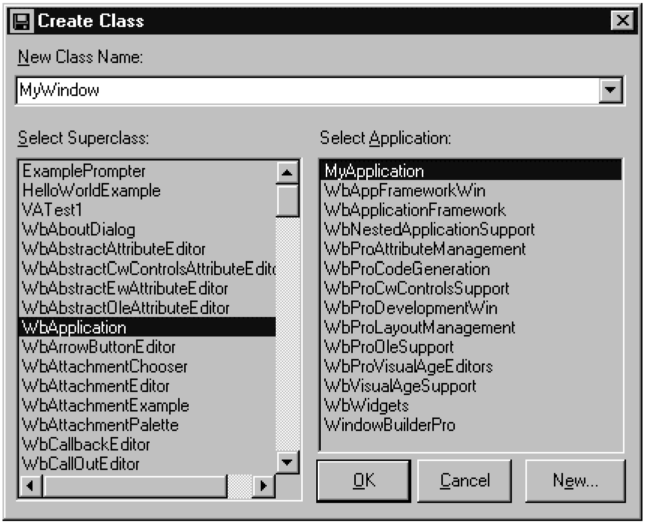
6. Click the **OK** button when you are done.

Before you can enter code for event handler methods, you must save the window. This allows WindowBuilder Pro to generate the stub (empty) event handler methods for you.

To save a window to disk:

1. Choose **Save** from the File menu or click the **Save** button in the toolbar

2. The Create Class dialog will appear.


Create Class

New Class Name: MyWindow

Select Superclass:
- ExamplePrompter
- HelloWorldExample
- VATest1
- WbAboutDialog
- WbAbstractAttributeEditor
- WbAbstractCwControlsAttributeEdit
- WbAbstractEwAttributeEditor
- WbAbstractOleAttributeEditor
- WbApplication
- WbArrowButtonEditor
- WbAttachmentChooser
- WbAttachmentEditor
- WbAttachmentExample
- WbAttachmentPalette
- WbCallbackEditor
- WbCallOutEditor

Select Application:
- MyApplication
- WbAppFrameworkWin
- WbApplicationFramework
- WbNestedApplicationSupport
- WbProAttributeManagement
- WbProCodeGeneration
- WbProCwControlsSupport
- WbProDevelopmentWin
- WbProLayoutManagement
- WbProOleSupport
- WbProVisualAgeEditors
- WbVisualAgeSupport
- WbWidgets
- WindowBuilderPro

3. Type the class name for the window in the **New Class Name** combo box.

4. Select a superclass from the **Select Superclass** list box.

5. Select an application in which you want the class to be created.

You must have an open edition of an application, or you will not be able to save the window. If you do not have an open edition, cancel the Create Class dialog, go to the Application Manager. Create either a new application, or a new edition of an existing application (the application must have WbApplicationFramework in its prerequisite chain). You can also click the **New** button to automatically create a new application with the selected superclass as a prerequisite.

6. Click the **OK** button to save the class and close the dialog.

You can enter code for event handler methods in any Class Browser. WindowBuilder Pro provides a button that brings up a browser on the class that you have created for the window.

To enter code for an event handler:

1. Click the Class Browser icon in the lower left of the screen. A browser for the subclass that you have created appears.

2. If the Public/Private button is displaying public, change it to private by clicking it.

By default, WindowBuilder Pro categorizes the callback methods it generates as private. This can be changed by using the Property Editor (accessible via the **Options | Properties** menu).

3. If you have method categories enabled, select the Callbacks category.

4. Locate and select the method corresponding to the method name that you entered in the **Method** combo box in the callback editor.

5. Type the code for the method.

Note that if long callback method annotations are enabled, the initial comment that WindowBuilder Pro generates for the method will describe all three of the arguments that are passed in to the method. Once you become more familiar with the system, you may want to turn off the long callback annotation feature via the Property Editor.

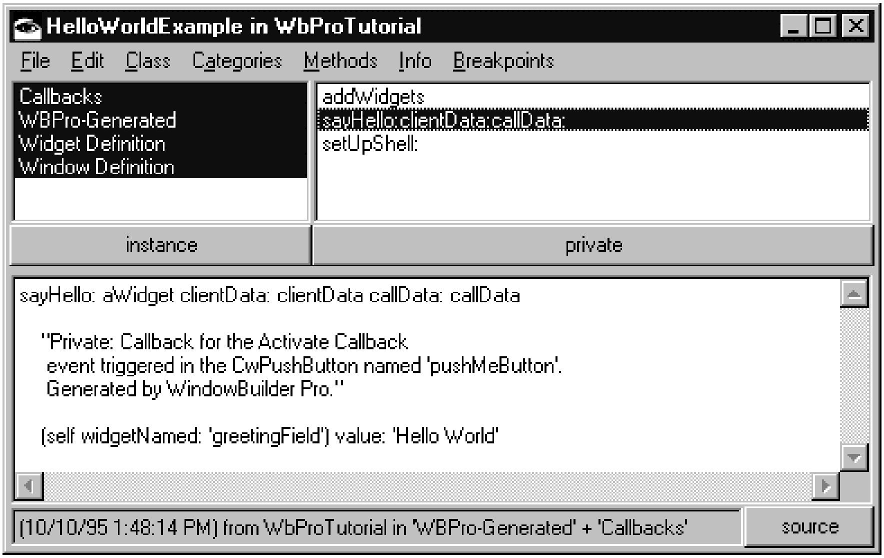
6. Save the method, and close the Class Browser when you are done with it.

Adding Callbacks and Event Handlers to the “Hello World” Application.

Now you are ready to enter the event handler code for the “Hello World” example application.

1. Select the push-button, and click the callback editor toolbar button. The Callback Editor appears.

2. Select the Activate callback in the Callback List.
3. Type “**sayHello::**” in the **Methods** combo box. Remember to include the two colons. The method name is automatically expanded to “sayHello:clientData:callData:”
4. Click **OK** to close the callback editor.
5. Save the window, and select the Class Browser button to bring up a browser.
6. Display the private methods for the Callbacks category. Select the `#sayHello:clientData:callData:` method, and enter the code for the method, as shown below.



```
sayHello: aWidget clientData: clientData callData: callData

   "Private: Callback for the Activate Callback
    event triggered in the CwPushButton named 'pushMeButton'.
    Generated by WindowBuilder Pro."

   (self widgetNamed: 'greetingField') value: 'Hello World'
```

(10/10/95 1:48:14 PM) from WbProTutorial in 'WBPro-Generated' + 'Callbacks'

7. Save the method, and close the Class Browser.

Testing an application

WindowBuilder Pro makes it easy to test your application. You can do so at any stage of the development process and switch between edit mode and testing at the click of a button.

To test an Application:

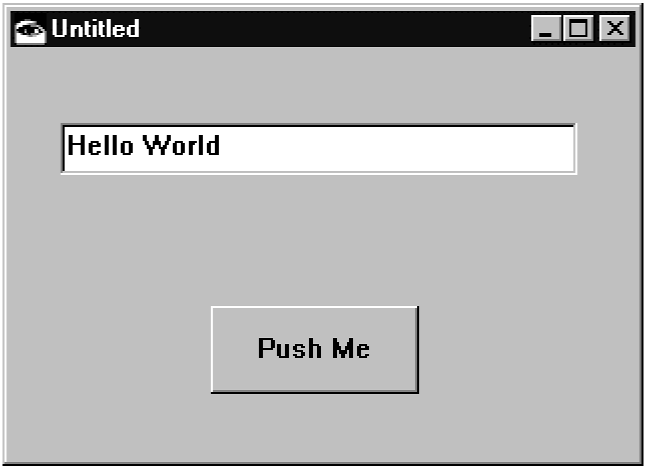
- From the toolbar, click the **Test Window** button.

Or, choose **Test Window** from the File menu.

Or, press CTRL+T.

Testing the “Hello World” Application

To test the “Hello World” application, click the **Test Window** button, or choose **Test Window** form the File menu. When the window appears, click the **Push Me** button. “Hello World” appears in the CwText widget, as shown below.



Hello World

“Hello World” application

Congratulations! You have completed your first WindowBuilder Pro application. Although this is a small example, you can see how WindowBuilder Pro makes creating windows-based applications simple. You also have an understanding of the general application-creation process. In the next two chapters, you’ll learn in detail about the features of WindowBuilder Pro.

Chapter 4 Using WindowBuilder Pro

In chapter 3, you learned that application development using WindowBuilder Pro consists of creating the user interface, and attaching callbacks to interface objects. In this chapter, you'll learn more about the tools Window Builder Pro provides to assist you in creating the user interface. Specifically, you'll learn about:

- Editing existing windows
- Positioning and sizing windows
- Positioning and sizing widgets
- Perform operations on multiple widgets
- Using the grid
- Setting the tab order for widgets
- Changing the fonts, colors, and styles of widgets
- Reframing widgets
- "Morphing" a widget from one type to another
- Factoring code using call-outs
- Using the Callback Editor
- Creating callbacks visually
- Managing Outboard Windows
- Using Popup Widget Menus

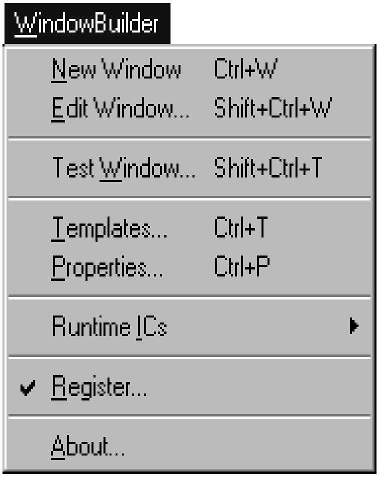
Editing Windows

You can easily edit any windows that you have created and saved with WindowBuilder Pro. WindowBuilder Pro generates all window definitions as subclasses of WbApplication. WbApplication is a powerful and flexible abstract superclass providing a generalized windowing framework (which is not found in the base image).

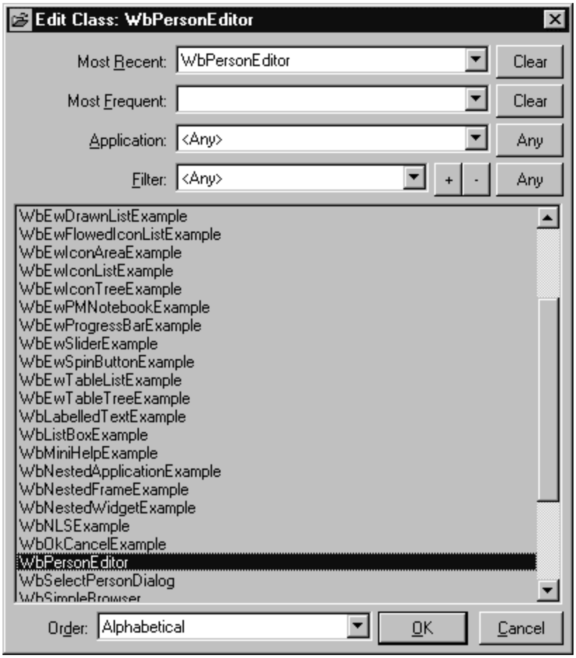
To edit an existing window:

1. Choose **Open** from the File menu or click the **Open** button in the toolbar.

Or, choose **Edit Window**... from the WindowBuilder Pro menu on the system Transcript.



2. The Edit Class dialog appears, as shown below.


Edit Class: WbPersonEditor

Most Recent: WbPersonEditor

Most Frequent:

Application: <Any>

Filter: <Any>

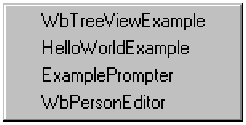
WbEwDrawnListExample
WbEwFlowedIconListExample
WbEwIconAreaExample
WbEwIconListExample
WbEwIconTreeExample
WbEwPMNotebookExample
WbEwProgressBarExample
WbEwSliderExample
WbEwSpinButtonExample
WbEwTableListExample
WbEwTableTreeExample
WbLabelledTextExample
WbListBoxExample
WbMiniHelpExample
WbNestedApplicationExample
WbNestedFrameExample
WbNestedWidgetExample
WbNLSExample
WbOkCancelExample
WbPersonEditor
WbSelectPersonDialog
WbSimpleBrowser

Order: Alphabetical

3. Select a window to edit.

The list in this dialog contains only those windows built by WindowBuilder Pro. The classes are listed alphabetically. They can also be viewed hierarchically or by frequency of access. A handy application filter is provided to help you look at just the classes that you wish. A user definable filter is also provided in which you can add an class you like (generally an abstract superclass for one or more of your own windows). As a further convenience, WindowBuilder Pro keeps track of the most recently accessed and most frequently accessed classes.

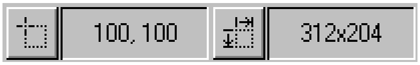
Note: Right-clicking on the **Open** button in the toolbar will popup a menu listing the most recently edited windows.



WbTreeViewExample
HelloWorldExample
ExamplePrompter
WbPersonEditor

Positioning and Sizing Windows

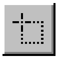
Two buttons are used to position and resize both windows and widgets. They are located on the status bar, as shown below.



100, 100
312x204

To set the initial window position:

1. Click the **Position** button.

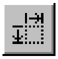


2. Type a point representing the desired upper left coordinate of the window.
3. Click **OK** to close the dialog. The window will reposition when you test the application.

The initial position of a window is its location on the screen at runtime.

To set the size of a window:

1. Click the **Window Size** button on the status bar.



2. Type a point representing the desired width and height of the window.
3. Click **OK** to close the dialog. The window resizes immediately.

The Position and Size icons can also be used to position and resize selected widgets.

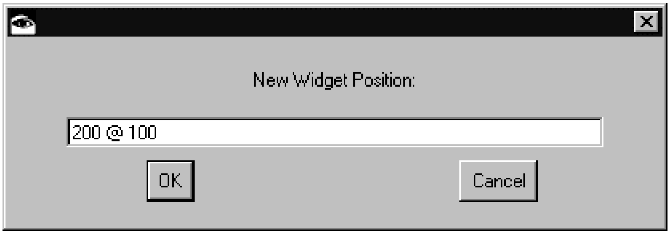
Positioning and Sizing Widgets

In the last chapter you learned how to position widgets by dragging and placing them in a window. There may be times when you need to position widgets at specific coordinates. You can enter coordinates directly by clicking the **Position** and **Size** buttons. You can also use these buttons to position and size windows. See the section, “Positioning and Sizing Windows.”

To position a widget:

1. Click the **Position** button, or choose **Set Widget Position** from the Size menu. A dialog appears, prompting you for the coordinates of the upper left corner of the widget.

2. Type the new coordinates.



New Widget Position:

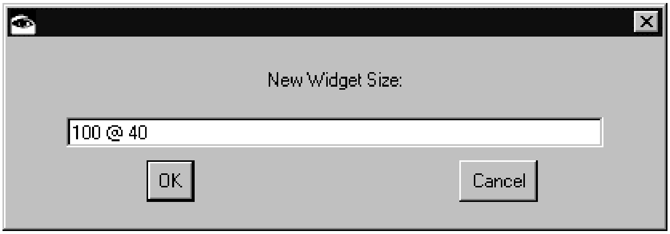
200 @ 100

3. Click **OK**. The widget immediately moves to the new position.

To size a widget:

1. Click the **Size** button, or choose **Set Widget Size** from the Size menu. A dialog appears, prompting you for the size of the widget.

2. Type the new coordinates.



New Widget Size:

100 @ 40

3. Click OK**.** The widget immediately resizes.

Remember that the size of the widget includes the widget's borders (if any).

Fine-tuning a Widget's Position

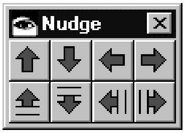
You may need to make fine adjustments to the position or size of a widget. These adjustments are available on the Position and Size menus, but are more easily made using their accelerator keys (for information on accelerator keys, see Chapter 5, *Menus*).

To move a widget in single-pixel increments:

1. Select the widget that you want to move.
2. Press CTRL and the arrow key corresponding to the direction toward which you want to resize.

Or, choose **Move By Pixel** from the Position menu. A submenu will appear, displaying a menu item for each direction.

Or, open up the floating Nudge tool via the **Options | Tools | Nudge** command and use the top row of buttons.



3. Choose the direction toward which you want to move the widget.

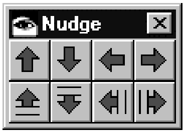
Resizing a widget a pixel at a time is a similar procedure.

To adjust the size of a widget by single pixel increments:

1. Select the widget that you want to resize.
2. Hold down the CTRL and SHIFT keys and press the arrow key corresponding to the direction toward which you want to resize.

Or, choose **Size By Pixel** from the Size menu. A submenu will appear, displaying a menu item for each direction.

Or, open up the floating Nudge tool via the **Options | Tools | Nudge** command and use the bottom row of buttons



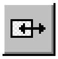
3. Choose the direction toward which you want to resize the widget.

Autosizing Widgets

When you add a text label to a widget, you can have the widget size around the text automatically, with a border. This is called autosizing. Autosizing sets the size of a widget to its preferred extent as defined by the widget itself. For example, for a CwLabel or CwPushButton widget, the size would be just large enough to fully contain the widget's label.

To Autosize a widget:

1. **Double-click** the widget you want to automatically size. The attribute editor for the widget appears.
2. In the **Label String** field, type the text label as you want it displayed.
3. Click **OK** to accept the change and close the attribute editor.
4. Choose **Auto Size Selection** from the Size menu, or click the **Autosize** button. The widget will size with a border around the text.



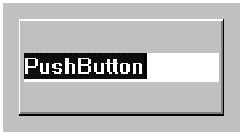
By default, autosizing is always on for instances of CwLabel and its subclasses (for example, CwPushButton). Instances of these classes automatically size around their labels. If you do not want a class of widgets to autosize, set the **Recompute Size** attribute of the widget class to false in the Template Editor. This will affect any new widgets of that class that you add to the edit window. Individual widgets can be told not to autosize by unchecking the **Recompute Size** checkbox in their attribute editors. You can still autosize a specific widget by typing its label in the widget's attribute editor and clicking the Autosize button.

Direct Editing

Many of the widgets that are editable by WindowBuilder Pro also support direct editing of their labels or contents.

To directly edit a widget:

1. Click the widget with the ALT key pressed. Either a single or multi-line edit region will appear, depending on the type of widget.



PushButton

2. Type the desired label.
3. Click anywhere outside of the edit region. The change takes effect immediately.

Using direct edit with a CwForm, CwRowColumn, CwFrame, WbFrame, CwScrolledWindow or any of the Notebook widgets allows you to manipulate the children of that widget. If the **Nested Direct Manipulation** option in the Options menu is enabled, you can directly click on a child widget and immediately put its parent(s) into direct edit mode. Children may be added to any of the above widgets by dropping the widget within the boundaries of the desired parent. Selecting multiple widgets follows the rule that only widgets at the same level (with the same parent) may be selected simultaneously.

If the **Allow Reparenting** option in the Options menu is enabled, you can drag widgets from one parent to another. For example, a widget may be dragged from the top level form into a nested form without the need to cut and paste it. Likewise, a table widget may be dragged into a scrolled window to make it scrollable.

Operations on Multiple Widgets

It is useful to be able to perform simple editing operations on groups of widgets, such as moving and deleting. This section covers how to create a group of widgets, and then discusses operations that are specific to widget groups.

Placing Multiple Widgets

You may need to place several widgets of the same type in a window. Using the left mouse button to place a widget unloads the widget from the cursor, requiring you to re-select the widget.

To place more than one of the same type of widget:

- Place the widget using the right mouse button. This will leave the cursor loaded with that widget, ready to place in another position.

Selecting Groups of Widgets

There are five way to create a group of widgets.

- The Select menu options: **Select All**, **Select All In Same Class**, **Select All In Same Hierarchy**.
- The **Select All** and **Select All In Same Class** buttons in the lower left corner of the WindowBuilder Pro window
- The Rubberband method (also known as the marquee selection method).
- The SHIFT-select method.
- The floating Widget Selection tool.

To select widgets using the Select menu options from the Edit menu:

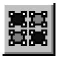
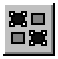
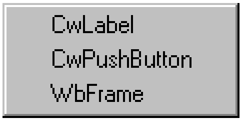
- Do one of the following:

To select	Do this
All of the widgets on the screen.	Choose **Select All**.
All widgets of the same type.	Select one widget of the desired type. Choose **Select All In Same Class**.
All widgets in the same class or subclass.	Select one widget of the common superclass. Choose **Select All In Same Hierarchy**.

The **Select All In Same Class** command is very useful in situations where you want to change an attribute of all widgets of a certain type that might be geographically dispersed around the screen.

To select widgets using the Select buttons:

- Do one of the following:

To select	Do this
All of the widgets on the screen.	Click the **Select All** button. 
All widgets of the same type.	Select one widget of the desired type. Click the **Select All In Same Class** button. 
All widgets of the same type.	**Right-click** on the **Select All** button. Select a widget type from the popup menu. 

To select widgets using the rubberband method:

1. Click and drag outside of the area containing the widgets you want to select. A selection box outline follows the mouse pointer.
2. Release the mouse button when all the widgets are enclosed within the selection box. The box disappears, and handles appear on all the widgets.

You are now able to perform operations on the widget group. Many operations that can be performed on individual widgets can be performed on groups, such as moving and deleting.

Notice that the handles of the highest widget in the z-order (the order of overlapping widgets is called the z-order) are solid black boxes, and the rest of the widgets' handles are outline boxes. This indicates that WindowBuilder Pro considers the widget with the black handles to be the first widget selected. This widget is called the model widget, for reasons that will become apparent in the discussion below.

Sometimes the highest widget in the z-order is not the widget you want to be the model. When you need to have more control specifying the model widget, choose the following method.

To select widgets using the SHIFT-select method:

- Select the first widget in the group. Holding down the SHIFT key, click the other widgets that you want in the group.

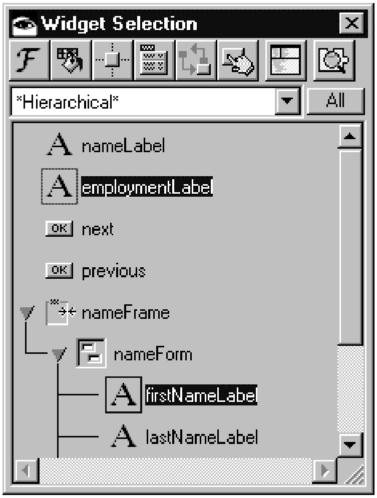
This method gives you complete control over which widget WindowBuilder Pro will consider the model widget for the group. This is important in the operations that follow.

If you prefer the VisualAge "target-is-last" mode of operation, select the Target Is First command from the Options menu. This will uncheck the command and reverse the selection order. Now the *last* widget selected will be the model. Although counter-intuitive at first, this method is useful in certain situations. For example, you can rubberband-select a group of widgets and then **SHIFT-deselect** and reselect the desired model widget.

To select widgets using the Widget Selection tool:

1. Choose the **Widget Selection** command from the Options | Tools menu. The Widget Selection tool appears as shown on the next page.
2. Select any of the widgets from the list.

The widget list can display the window's widgets either hierarchically or alphabetically by name or by type. It can also filter the list to show only one type of widget at a time. The **All** button in the upper right corner makes it easy to select all of the widgets in the window or all the widgets of a particular type.



Replicating Widget Sizes

Often you need to make widgets the same size. You can either make one widget and make multiple copies, or use the **Replicate** buttons.

To replicate widget sizes:

1. Create the desired widgets.
2. Set one of the widgets to the desired size, and leave it selected. This is the model for the other widgets.
3. **SHIFT-select** the remaining widgets.
4. Choose **Replicate Width** from the Size menu, or the **Replicate Width** button.

The widgets immediately assume the width of the model widget.

5. Choose **Replicate Height** from the Size menu, or the **Replicate Height** button.

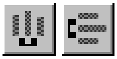
The widgets immediately assume the height of the model widget.

Distributing Widgets

Placing widgets exactly equal distances from each other in a window can be a time consuming process. WindowBuilder Pro requires only that you establish the positions of the end widgets, and then automatically distributes the rest evenly between the end widgets.

To distribute widgets:

1. Position the two endpoint widgets where you want the ends of the row or column to be. Leave one of the endpoint widgets selected.
2. **SHIFT-select** the rest of the widgets to be distributed. Make sure to select the other endpoint last.
3. Choose **Distribute Horizontally** or **Distribute Vertically** from the Position menu, or click one of the **Distribute** buttons.

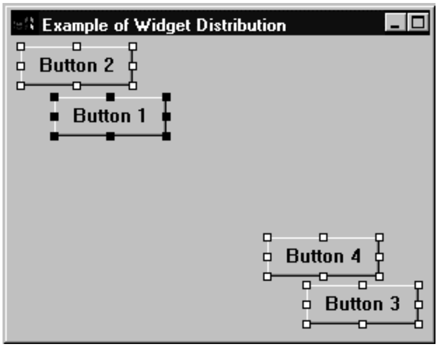


4. Choose the direction in which you want to distribute the widgets. The widgets immediately distribute evenly between the two endpoint widgets.

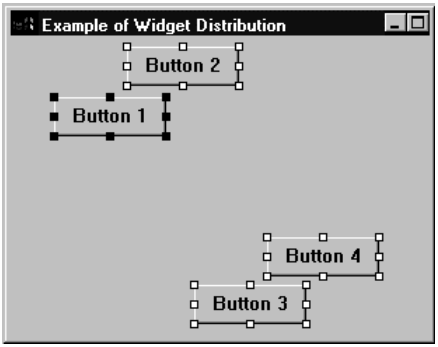
The key to this operation is positioning the endpoint widgets. Note that WindowBuilder Pro distributes the other widgets equally in the space between the endpoint widgets, not in the space between the edges of the window. Note also that the widgets don’t need to be aligned for distribution to work properly.

If the ALT key is held down while performing a horizontal or vertical widget distribution, the widgets will be distributed based on their relative position rather than the order in which the were selected.

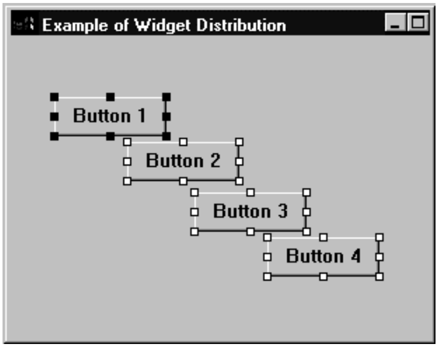
In the example shown below, the four buttons have been selected in numeric order with Button 1 first, and Button 4 last. Selecting **Distribute** distributes them, resulting in the window shown in the third picture.



Buttons, before vertical Distribute operation. Button 1 and Button 4 are in desired vertical position. Button 1 has been selected first, Button 4 has been selected last.



Button 2 and Button 3 have been moved to the desired horizontal position in preparation for the Distribute operation.



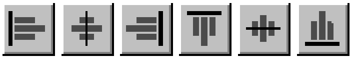
Final position of buttons, after Distribute operation.

Aligning Widgets

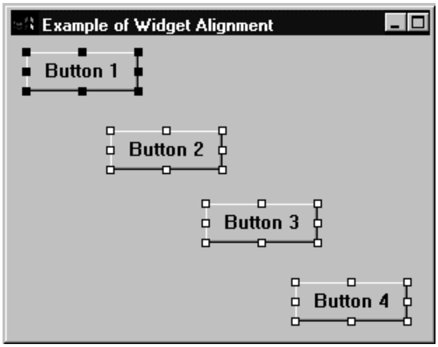
Precise alignment of rows of widgets is important to the look of an application. It can be difficult to manually arrange a row of widgets so they align along one edge. WindowBuilder Pro provides tools that automatically align widgets along their tops, bottoms, or sides.

To align widgets:

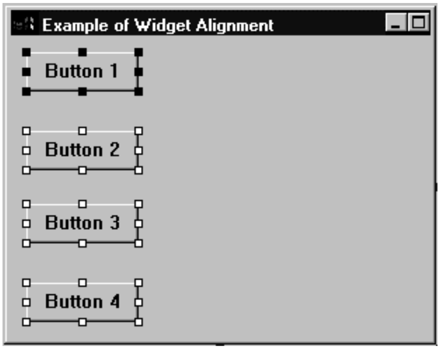
1. Move one of the widgets to the correct horizontal or vertical position, and leave it selected. This is the model widget for the alignment operation.
2. **SHIFT-select** the rest of the widgets that you want to align.
3. Click the button corresponding to the side of the widgets that you want aligned. The widgets align immediately.



A button is available for each choice. These choices are also available from the Align menu.



Buttons, ready to be aligned. Button 1 is the model widget.



Buttons after alignment.

Note: If the ALT key is held down while performing a horizontal or vertical widget alignment, only the specified sides of the widgets will be aligned while the opposite sides will not move. This will cause the widgets to grow or shrink in size (as opposed to moving and retaining their original sizes)

Using the Grid

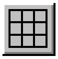
WindowBuilder Pro has a snap-to-grid feature. When widgets are placed, sized and moved, they snap to the hidden grid. By default, the distance (in pixels) between any two grid lines is 4 @ 4. You can change it to any size you want by using the **Set Grid Size** on the Options menu or via the popup menu on the **Grid** button. You can turn the grid off by unchecking **Use Grid** on the Options menu. WindowBuilder Pro saves the setting between sessions.

To change the grid size via the Set Grid Size command:

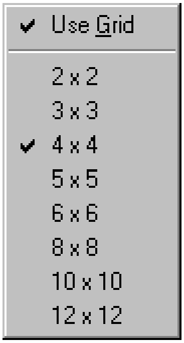
1. Choose **Set Grid Size** from the Options menu, or **right-click** the **Grid** button. A dialog appears, displaying a point value.
2. Type the new point value, where the x value represents the number of pixels between vertical grid lines, and the y value is the number of pixels between horizontal grid lines.
3. Click **OK** to close the dialog.

To change the grid size using the Grid button's popup menu:

1. **Right-click** on the **Grid** button in the lower right corner of the WindowBuilder Pro window.



2. Select an appropriate grid size from the popup menu.



To display the grid:

- Choose **Draw Grid** from the Options menu, or click the **Grid** button.

Setting the Tab Order for Widgets

It is important to consider users who prefer to navigate the application by using the keyboard, rather than the mouse. You can allow users to use the tab key to move from widget to widget. To do so, you must set the order in which the widgets are accessed when the user presses the tab key. By default, the tab order is the order in which the widgets are physically added to the screen. In other words, the first widget added to the screen is considered the topmost widget in the z-order. WindowBuilder Pro provides a powerful, yet easy to use, tab order editor to manipulate this order.

To see the current tab and z-order:

- Choose **Show Tab & Z-Order** from the Options menu.

  Or, **ALT-click** the **Tab & Z-Order** button in the lower left corner of the WindowBuilder Pro window

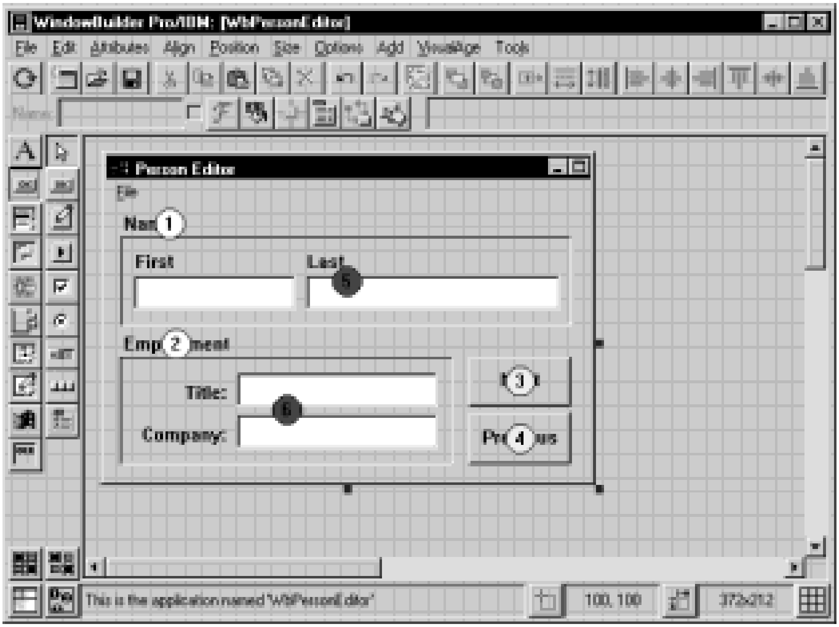
  A round, color-coded number appears on each widget.

The number indicates the widget’s position in the z-order. The color of the circle indicates the widget’s status as a tab stop.

Red indicates widgets that are both tab stops and tab groups (for example, full tab stops).

Yellow indicates widgets that are tab stops but not groups (for example, buttons in a tab group).

White indicates widgets that are not tab stops (for example, labels and separators).

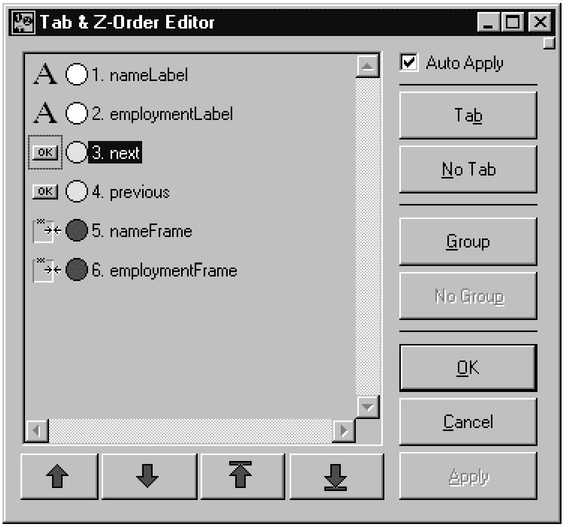

WindowBuilder Pro/IBM: [WbPersonEditor]
File Edit Attributes Align Position Size Options Add VisualAge Tools
Name:
Person Editor
File
Nam 1
First
Last 5
Emp 2 ment
Title:
Company:
6
3
Pr 4 us
100, 100
372x212

This is the application named 'WbPersonEditor'

To set the tab order via the Tab & Z-Order Editor:

1. Click the **Tab & Z-Order** button, or choose **Tab & Z-Order** from the Attributes menu.

The Tab & Z-Order Editor appears, as shown below



☑ Auto Apply

○ 1. nameLabel
○ 2. employmentLabel
○ 3. next
○ 4. previous
● 5. nameFrame
● 6. employmentFrame

2. Select the widget or widgets that you want to reposition.
3. Click either the ⬆, ⬇, ⤒ or ⤓ buttons to change the order of the selected widgets. If a discontinuous set of widgets is selected, they will all be collected together and follow the movement of the first widget in the selection.
4. Click either the **Tab** or **No Tab** buttons to specify whether the selected widgets are to be tab stops or not. Full tab stops are indicated by a red circle next to the name of the widget.
5. Click either the **Group** or **No Group** buttons to specify whether the selected widgets are tab groups as well. Widgets that are not tab groups become part of a single general tab group (this is primarily useful for button widgets). Widgets that are tab stops but not tab groups are indicated by a yellow circle.

To set the tab order via drag and drop:

1. Choose **Show Tab & Z-Order** from the Options menu.

Or, **ALT-click** the **Tab & Z-Order** button in the lower left corner of the WindowBuilder Pro window

2. Drag the color-coded tags from one widget to another.

To make a widget first in the tab order, find the tag labeled “1” and drag it to the desired widget. It will now become the first widget in the tab order (the widget that originally held that position will now become the second widget in the tab order).

Re-ordering Widgets

When you select widgets in the Tab & Z-Order Editor, they are also selected in the design area of the main editing window. Conversely, when you select widgets in the main editing window they are selected in the Tab & Z-Order Editor as well.

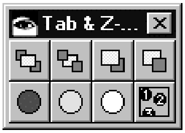
To re-order all of the widgets on the screen (or any subset):

1. **SHIFT-select** the widgets in the desired order. As you select them, they become selected in the Tab & Z-Order editor.
2. Click either the ⬆, ⬇, ⤒ or ⤓ button (at least one will be available) in the Tab & Z-Order Editor. The selected widgets re-order immediately in the order in which you selected them.
3. **CTRL-clicking** on the **Tab & Z-Order** button in the main editor will also cause the widgets to re-order.

The initial, empty window is the top level. **ALT-clicking** to direct edit a CwForm or CwRowColumn widget moves you down a level (in other words, all editing functions apply to that CwForm or CwRowColumn). The Tab & Z-Order Editor only displays widgets at the current editing level. The **Show Tab & Z-Order** command only applies to the current editing level.

Pay special attention to the interaction between tab stops, tab groups and the initial tab stop. The initial focus goes to the first widget that is a tab stop and is not a tab group (for example, a button that is part of the top level tab group). If all widgets are tab groups, focus goes to the first widget that is a full tab stop (i.e., both a tab stop and a tab group, like a CwText). Widgets which do not receive focus are not part of the tab order, whereas they are part of the z-order (for example, CwLabels and CwSeparators).

Note: WindowBuilder Pro also provides a floating tab & z-order tool. This tool is used for fine-tuning the tab and z-order of the selected widgets without the need to open the Tab & Z-Order Editor. Access this tool via the **Options | Tools | Tab & Z-Order** command.



Changing Fonts

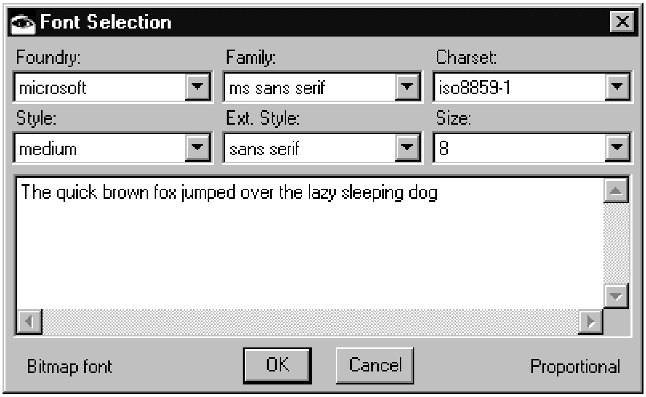
By default, the standard system font is used for all widgets. You can change the font with the Font Editor.

To change the font:

1. Select the widget or widgets whose font you wish to change.
2. Click the **Font** button, or choose **Font** from the Attributes menu.

The standard VisualAge font dialog will appear, as shown on the next page.

3. Specify the **Family**, **Style** and **Size** of the desired font.
4. Click **OK** to confirm the change.



Foundry: microsoft
Family: ms sans serif
Charset: iso8859-1
Style: medium
Ext. Style: sans serif
Size: 8

The quick brown fox jumped over the lazy sleeping dog

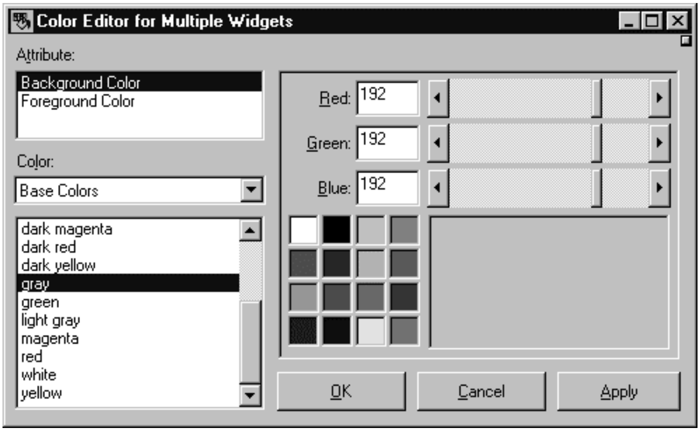
Setting Colors

Each type of widget in the system has a default foreground and background color associated with it. The Color Editor provides a means to change these colors and any other color attributes a widget might have.

To set the color:

1. Select the widget or widgets whose colors you wish to change.
2. Click the **Color** button, or choose **Color** from the Attributes menu.

The Color Editor appears, as shown below.



Attribute:
Background Color
Foreground Color

Color: Base Colors

dark magenta
dark red
dark yellow
gray
green
light gray
magenta
red
white
yellow

Red: 192
Green: 192
Blue: 192

3. Select the color attribute that you wish to change (generally just “Background Color” and “Foreground” color). When a color attribute is selected, the name of the color

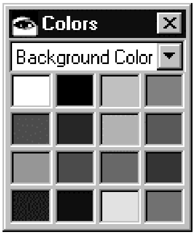
will be selected in the color list and the RGB values and a rendering of the color will appear in the RGB color editor on the right.

4. Either select the desired color from the color list, color palette, or use the RGB color editor's scrollbars or entry fields to set the red, green and blue values of the desired color. When you set one of the RGB values, the system selects the closest matching color from the color list for you.
5. Click **OK** to confirm the color choice.

The Color Editor provides three different color lists: “Base Color”, “R3 Color” and “R4 Color”. The first list provides the standard 16 colors supported by the default palettes on all platforms. The other lists are colors supported under various versions of UNIX (for example, R3 & R4). The VisualAge color model, like the windowing system, is based on UNIX Motif. The list of supported colors varies by platform and video driver capabilities. If you select a color that is not in the Base Color list, the operating system tries to match that color to the best of its ability. Windows responds to a request for an unsupported color by displaying the closest color from its default palette. OS/2 responds by dithering the colors that it is capable of displaying.

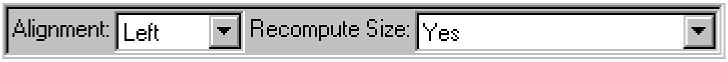
The first item in the color list will always be “<Default>”. Selecting this color will set the widget to use its own default colors for that attribute. When a widget has been defined to use its own default color, no color attribute code will be generated.

Note: WindowBuilder Pro also provides a floating color tool. This is tool can be used to set the colors for the selected widgets without the need to open the Color Editor. Access this tool via the **Options | Tools | Color** command.



Styles

Each widget available to WindowBuilder Pro supports many attributes. One or more of these attributes represent the major style of the widget. For example, Left, Right or Center aligned is the major style for CwLabels. While the attribute editors for each widget provide a means for changing this style information, WindowBuilder Pro also provides a fast path for setting the major style attributes of the widget.



Alignment: Left

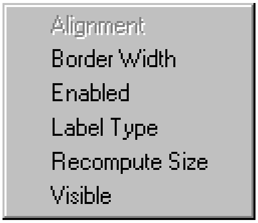
Recompute Size: Yes

The style comboboxes update their styles to reflect the currently selected widgets in the design surface. Note that the selected widgets do not need to be homogeneous. If a heterogeneous collection of widgets is selected, the two style comboboxes will reflect attributes that all of the selected widgets have in common (e.g., enable/disable). If all of the selected widgets have the same value for the displayed attribute, this value will be shown in the combobox. If they have differing values, the combobox value will be empty. In any case, selecting a value from the combobox will assign that attribute setting to all of the selected widgets

To change the style:

1. Select the widget or widgets for which you wish to change the style. The contents of the two Style combo boxes update to reflect the style options for the selected widgets.
2. Select the desired style. Visible changes, if any, will be reflected immediately.

The style comboboxes are completely customizable. **Right-click** on either of them to get a popup menu of other style attributes specific to the selected widgets. Selecting a new attribute will update the combobox's label and contents. If the first style combobox is changed, the second will adopt the style attribute formerly held by the first. Think of the two style comboboxes as showing you a window on a collection of attributes where the most important two will be displayed. By changing the attribute settings, you are essentially re-ordering the list and telling WindowBuilder Pro what you think the most important attributes are. Any changes you make will be remembered so that the next time you select a widget of the same type, the two comboboxes will reflect the last settings that you made.



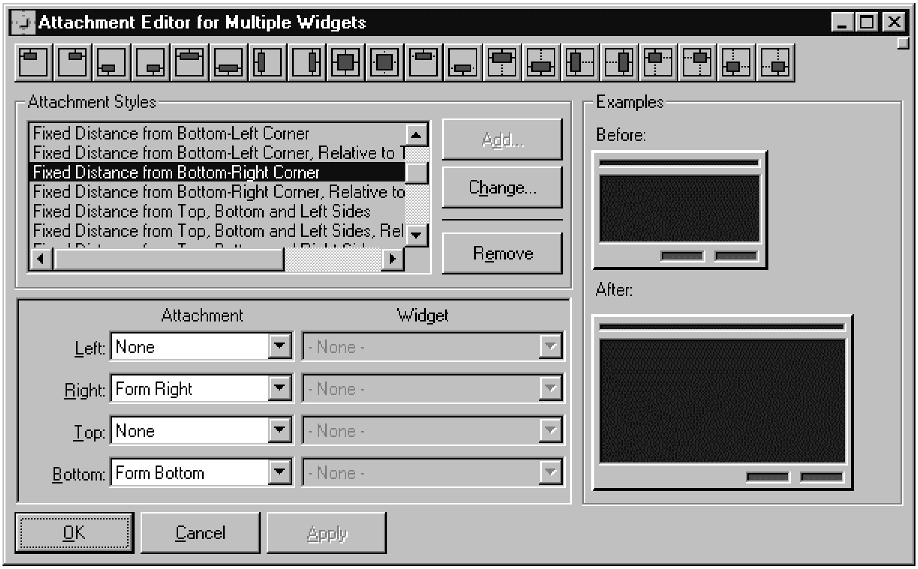
Reframing Widgets

Most applications that you create in WindowBuilder Pro will have resizeable windows. When a window resizes, the widgets within it usually resize or move as well. In VisualAge, this is accomplished by specifying the widget's attachments. All kinds of attachments may be specified: attachments with respect to the application window; attachments with respect to any other widget in the form. Proportional attachments and fixed size attachments can also be specified.

To set the attachments:

1. Select the widget or widgets whose attachments you wish to change.
2. Click the **Attachment** button, or choose the **Attachments** command from the Attributes menu.

The Attachment Editor appears, as shown on the next page.



Attachment Editor for Multiple Widgets

Attachment Styles

Fixed Distance from Bottom-Left Corner
Fixed Distance from Bottom-Left Corner, Relative to T
Fixed Distance from Bottom-Right Corner
Fixed Distance from Bottom-Right Corner, Relative to
Fixed Distance from Top, Bottom and Left Sides
Fixed Distance from Top, Bottom and Left Sides, Rel

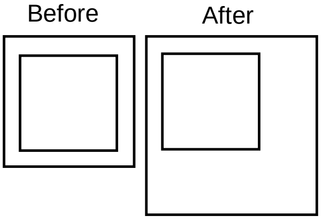
	Attachment	Widget
Left:	None	- None -
Right:	Form Right	- None -
Top:	None	- None -
Bottom:	Form Bottom	- None -

3. Set the desired attachment type for each of the widget's four sides. For the left side, you can set whether the coordinate should always be a fixed distance from the form's left side, the form's right side, a fixed relative to its own right side (None), relative (proportional) to its initial position on the form, or fixed relative to the right side of any other widget.
4. Click **OK** or **Apply** to confirm the attachment choices.

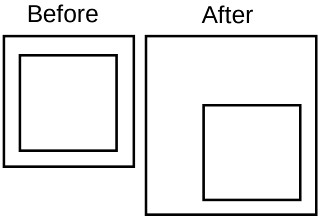
At least one side in both the vertical and horizontal directions must be specified. If, for example, a widget's left side is fixed to its own right side, the attachments for the widget's right side must be fixed somehow to its form; otherwise you'd have a pretty confused widget floating around!

As an alternative to fixing the left side to something, you can also set it proportionally. In this case, the left side will always be in the same proportional position within the window, no matter how large or small you size it.

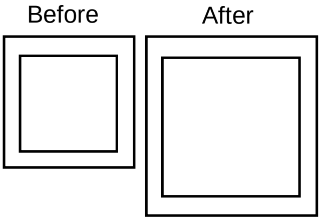
As mentioned above, this discussion holds true for all four sides of a widget. Since each side can be specified separately from all other sides, you can create many different possible variations, covering most common resizing situations. To help illustrate the possibilities, we've provided the following examples:



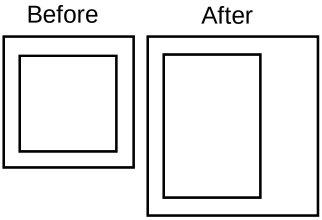
Fixed to upper left corner
(the default)



Fixed to lower right corner



Fixed on all sides



Fixed on top, bottom and
left sides

To set the attachments for multiple widgets:

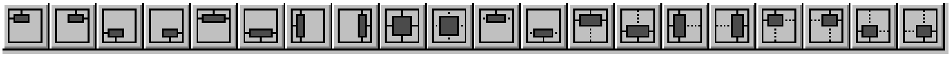
1. Multiple-select the desired widgets.
2. Click the **Attachments** button.

For more information on multiple selection, see Selecting Multiple Widgets.

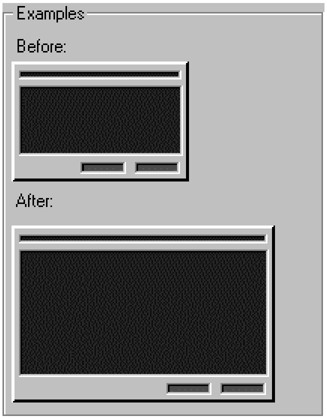
If the **Update Outboards** option is on, the Attachment Editor may be left open. As widgets are selected in the main editor, their current settings will be reflected in the

Attachment Editor. They may then be changed, and those changes locked in, by clicking the **Apply** button.

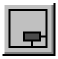
Attachment Styles



Many different attachment combinations can be specified. However, you will probably only use a few of the possible combinations on a frequent basis. Recognizing this, the Attachment Editor provides a fast path to the common attachment combinations, or *styles*. The toolbar and listbox at the top of the Attachment Editor provide access to all of the most common attachment styles. As you select a style, the individual setting for each side are reflected in the combo boxes below and the thumb-nail view on the right side of the window.



Locking a button to the lower right corner of the window—an operation that would normally take four steps—can be accomplished in one step by selecting the “Fixed Distance from Bottom-Right Corner” style or selecting the appropriate toolbar button.

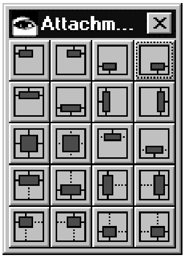


To Add a new attachment style:

1. Set the desired combination of styles in the combo boxes at the bottom of the screen. If you select a combination that is not already defined, the **Add** button will become enabled.
2. Click the **Add** button.
3. Enter the name for the new style in the prompter that appears.
4. Click **OK** to save your style.

Once you have saved a style, WindowBuilder Pro remembers it for the future. The names of existing styles may be changed by clicking the **Change** button. Styles that are not needed may be deleted by clicking the **Remove** button.

Note: WindowBuilder Pro also provides a floating attachment tool. This is tool can be used to set the attachments for the selected widgets without the need to open the Attachment Editor. Each button in the tool represents a different attachment style. Access this tool via the **Options | Tools | Attachments** command.

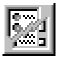


Widget Morphing

Morphing allows you to quickly change any widget from one type to another, allowing for powerful “what-if” style visual development. For example, a WbScrolledList instance could be converted into a CwComboBox or WbRadioBox. Common attributes are automatically translated. Attributes not needed by the target class are lost. Attributes not provided by the source class are defaulted.

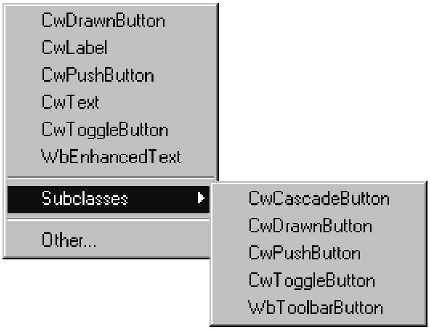
To morph a widget:

1. Select the widget or widgets that you wish to morph.
2. Select **Morph**... from the Attributes menu or click the **Morph** button.



3. Select the new widget class from the list of widget types presented.
4. Click **OK** to morph the widget into the new type.

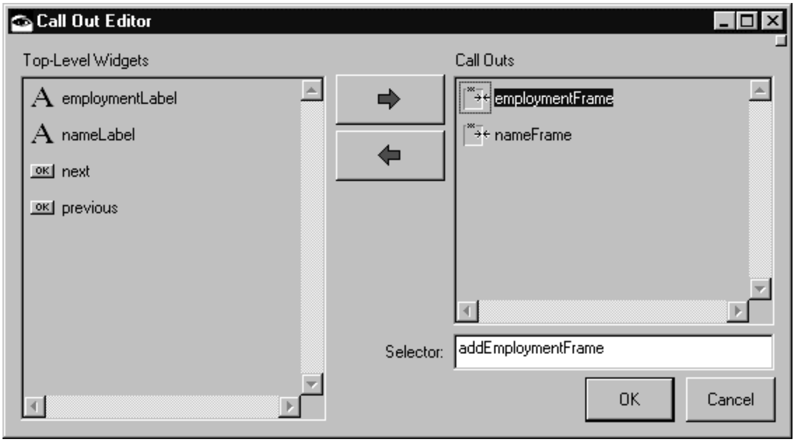
Right-clicking on the **Morph** button in the toolbar will popup a menu containing all of the lists morphing types for the selected widget. If a widget class has subclasses, all of those are listed as well.



Warning: Some care is needed on your part when morphing a widget into a radically different type. WindowBuilder Pro maps over any attributes the two widgets have in common as well as event callbacks for any shared events. You must be careful that the event callbacks do, in fact, make sense for the new type. For example, a Default Action Callback for a CwText would not be appropriate for an CwPushButton. It is recommended that morphing be limited to similar classes of objects.

Using Call Outs

Call Outs give you the opportunity to exercise more control over how your window definition code is factored. For a large window with lots of widget definitions, WindowBuilder Pro will generate a very large `#addWidgets` method. The Call Out Editor allows you to have any top level widget (e.g., child of the main form) generated into its own method. If a CwForm, CwRowColumn, CwScrolledWindow, Frame or Notebook widget is so designated, it and all of its nested children will be generated in the specified method.



Selector: addEmploymentFrame

To create a call out:

1. Open the Call Out Editor.
2. Select a top level widget in the left hand list for which you wish to add a call out.
3. Click the ⇨ button to move the widget to the right hand list.
4. Optionally modify the automatically created call out selector. Note that this should be unary selector (no arguments).
5. Click **OK** to confirm the call out specifications.

To remove a call out:

1. Open the Call Out Editor.
2. Select the top level widget in the right hand list from which you wish to remove the call out.
3. Click the ⇦ button to move the widget to the left hand list.

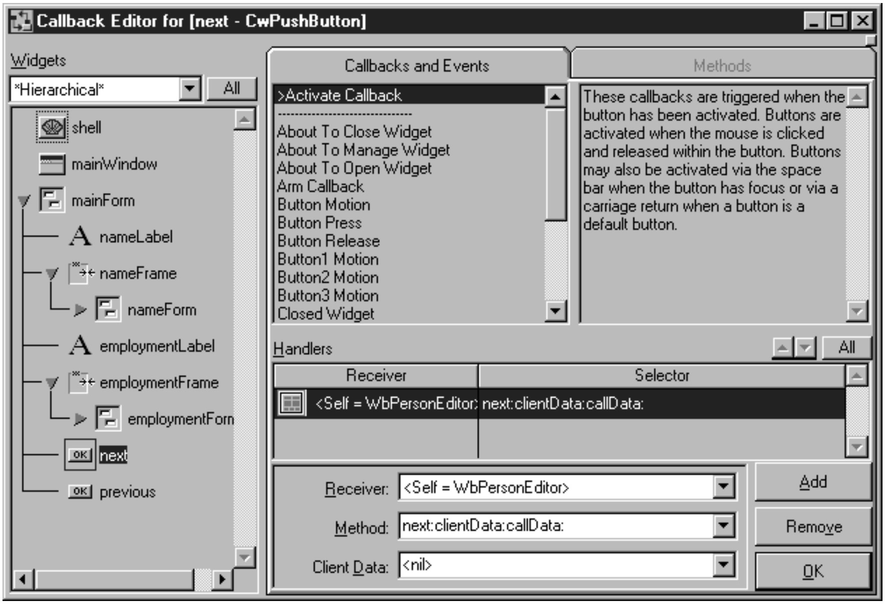
Using the Callback Editor

Actions performed on widgets by the user must be communicated back to the application. One mechanism used for this communication is a *callback*. A **callback method** defines actions to perform in response to some occurrence in a widget. Callbacks are normally registered just after widgets are created. For example, when a push button widget is created, the application usually registers an **Activate** callback that is executed when the button is activated by the user clicking on it. Although it is not necessary for the application to register callbacks, without them the application is unable to take action based on the user's interaction with the widgets.

To define a callback or event handler for a widget:

1. Select the widget or widget to which you would like to register a callback
2. Click the **Callback** button, or choose **Callbacks** from the Attributes menu.

The Callback Editor will appear as shown on the next page:



These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Receiver	Selector
<Self = WbPersonEditor>	next:clientData:callData:

Receiver: <Self = WbPersonEditor>

Method: next:clientData:callData:

Client Data: <nil>

3. Select the type of callback or event to which the application should respond (a description of the callback or event appears to the right). As a convenience, the most common callbacks for a widget appear first in the list.

4. Click the **Add** button or double-click the callback type. A new handler appears in the Handlers list.

5. Specify the receiver of the callback by selecting an item from the **Receiver** list. Generally, the receiver is the application itself ("self").

6. Type the name of the method in the **Method**: combo box, followed by two colons. When you type the second colon, the editor automatically fills in the rest of the method name by appending `clientData:callData`: for you.

Zero (unary) and single argument callback handlers are also supported. Leave the second colon off, if you do not want the method name expanded to the standard three argument style.

7. Type client data, if any, in the **Client Data** combo box.

8. Click the **OK** button when you are done.

To remove one or more callback handlers:

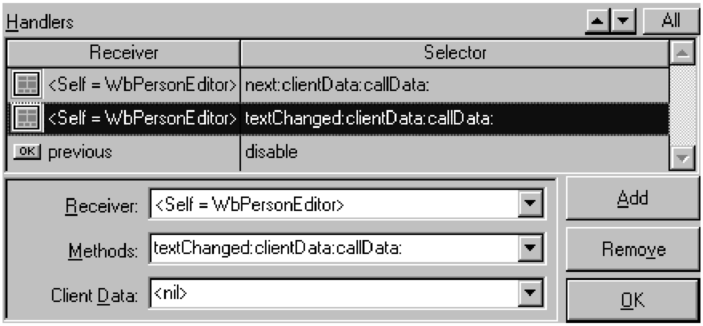
1. Select the callback handlers in the handlers list that you would like to remove.

2. Click the **Remove** button in the lower right corner of the editor.

By default, multiple callback handlers for the same callback are executed in the order they are defined. The handlers may be reordered if necessary.

To re-order callback handlers:

1. Select the callback handler in the handlers list that you would like to re-order.



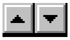
Receiver	Selector
<Self = WbPersonEditor>	next:clientData:callData:
<Self = WbPersonEditor>	textChanged:clientData:callData:
previous	disable

Receiver: <Self = WbPersonEditor>

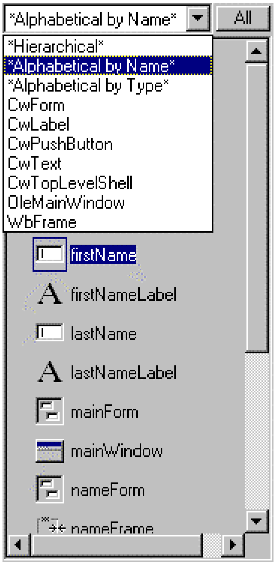
Methods: textChanged:clientData:callData:

Client Data: <nil>

2. Click the **Up** or **Down** button as appropriate.

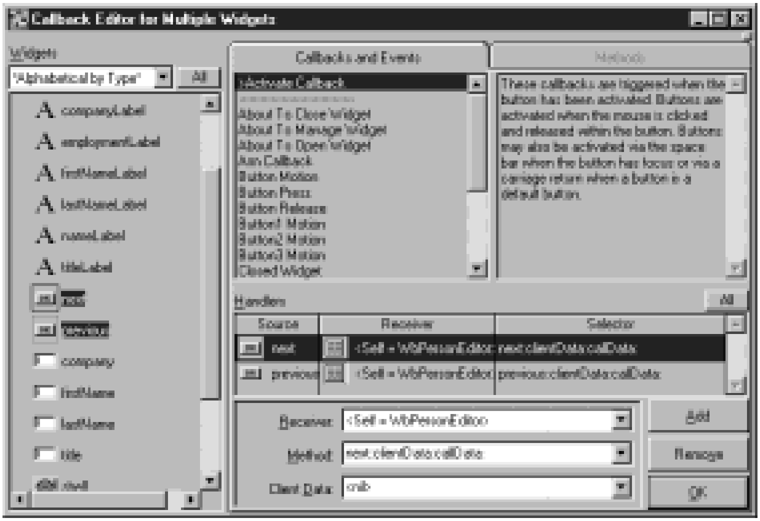


The widget list on the left-hand side of the Callback Editor can display the window’s widgets either hierarchically or alphabetically by name or by type. It can also filter the list to show only one type of widget at a time. The **All** button in the upper right corner makes it easy to select all of the widgets in the window or all the widgets of a particular type.



Multiple callback handlers may be defined for multiple widgets simultaneously. This would be the case if you wanted several widgets to respond to the same callback or event in the same way (in order to update a context sensitive help system, for example).

If multiple widgets are selected within the Callback Editor, the **Callback and Event** list will show the intersection of the callbacks that the widgets have in common. If a callback or event is selected, the **Handlers** list will show all handlers for that callback or event defined in all of the selected widgets. The handlers list will display a third column, **Source**, in order to identify the source widget for each handler. Since the **Up** and **Down** buttons only apply when the displayed handlers are for a single widget, they are not displayed.



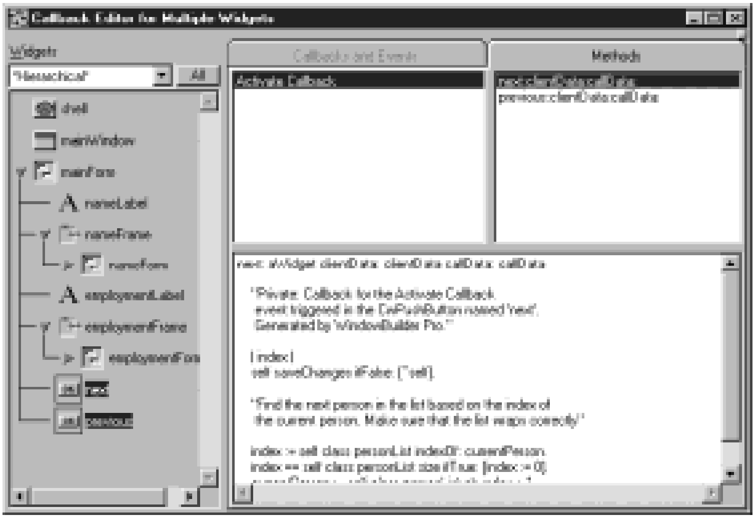
These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Source	Receiver	Selector
next	<Self = WbPersonEditor>	next:clientData:callData:
previous	<Self = WbPersonEditor>	previous:clientData:callData:

Receiver: <Self = WbPersonEditor>
Method: next:clientData:callData:
Client Data: <nil>

Multiple handlers may also be selected and edited simultaneously. When multiple handlers are selected, changes made in the **Receiver**, **Method** or **Client Data** fields are immediately applied to all of the selected handlers. This makes it easy to configure the same handler for multiple widgets. The **All** button above the handlers list provides a convenient way to select all of the handlers without scrolling.

The callback handlers themselves may be viewed and edited using the embedded code browser. Just click on the **Methods** tab to access the embedded code browser. Click on any widget or group of widgets and the list of callbacks that have handlers will be shown in the upper left list. Select one or more callbacks and view the callbacks handlers that have been defined for them. Clicking on any of the callback handlers (methods) will display its code in the text pane at the bottom of the window. **Right-clicking** on either the methods list or the text pane will popup the standard menus that you would expect to find in any of the system code browsers. Code may be viewed and edited here. Methods may be deleted, added, new editionss loaded, etc.



```
next: aWidget clientData: clientData callData: callData

  "Private: Callback for the Activate Callback
   event triggered in the CwPushButton named 'next'.
   Generated by 'WindowBuilder Pro.'"

  | index |
  self saveChanges ifFalse: [^self].

  "Find the next person in the list based on the index of
   the current person. Make sure that the list wraps correctly."

  index := self class personList indexOf: currentPerson.
  index == self class personList size ifTrue: [index := 0].
```

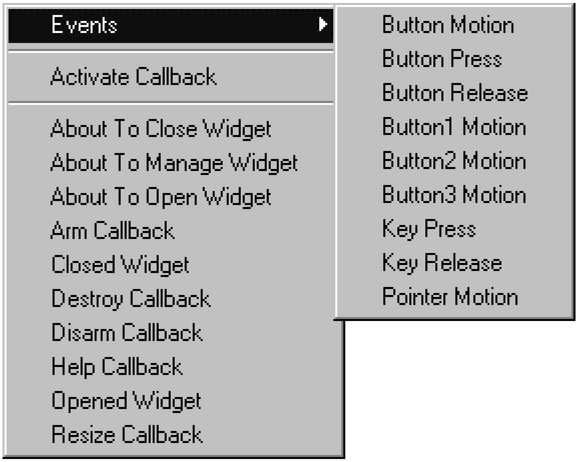
Creating Callback Handlers Visually

New with version 4.0, WindowBuilder Pro gives you the ability to define callback and event handlers visually by hooking widgets together - much like the VisualAge Composition Editor. This “visual shorthand” provides an alternative to defining callback and event handlers via the Callback Editor.

Note that this feature does not provide support for complete visual programming. It only applies to the window and widgets currently being edited. Non-visual objects are not yet supported. Also, it does not *yet* provide a persistent view of visual connections. Once a connection is established (which is simply a shorthand approach to creating a callback handler), it can only be edited (e.g., deleted, receiver or selector modified, etc.) in the Callback Editor.

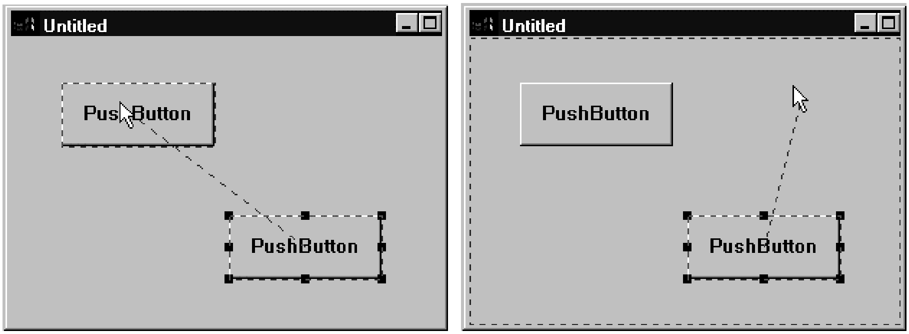
To create a callback handler visually:

1. Select the widget that will be the initiator of the event.
2. Right-click on the widget to reveal the popup widget menu and select the Connect submenu, or **ALT-right-click** on the widget right-click on the Callback button to popup the Connect menu.

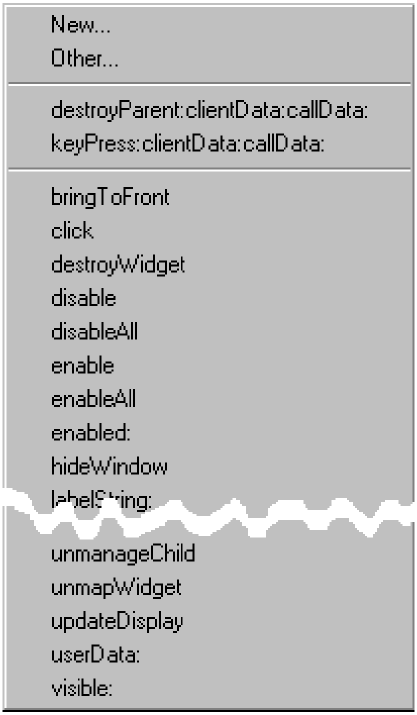


3. Select a callback or event. For convenience, the most important callbacks triggered by the widget are listed first. All events are listed in a submenu called Events.

4. The cursor will enter visual connection mode. Move the mouse pointer to the desired target of the connection. This can be one of the other widgets or the application itself. As you move the mouse pointer over various targets, they will highlight as shown below.



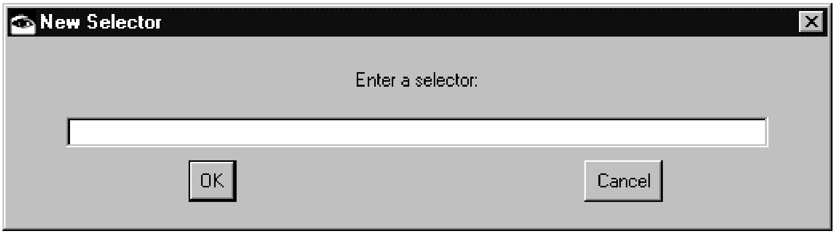
5. Release the mouse button when it is over the desired target. A popup menu will appear listing all of the target's messages that can be connected to.



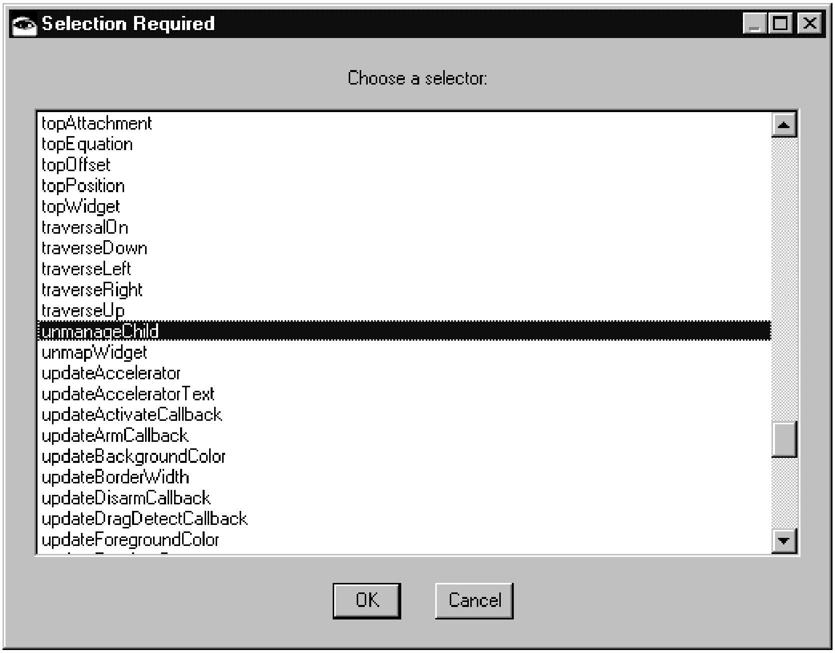
6. Select the desired target message. Standard VisualAge three-argument callbacks (listed first) are supported as well as zero (unary) and one-argument callbacks.

   For one-argument callback handlers, the argument that is passed is automatically determined based upon the source and type of the callback. For example, if the source widget is a listbox and the Selection Callback has been chosen, the default argument is the selected item of the listbox. The argument passed by the callback can be changed via the Callback Editor.

7. In addition to the listed messages, you can link to a new message or any other message supported by the target. If **New** is selected, the dialog shown below is displayed. A zero, one or three-argument selector may be specified (for a standard three-argument selector, just type the first element of the selector followed by two colons).



If **Other** is selected, the dialog show below is displayed. A selector may be chosen from the list.



topAttachment
topEquation
topOffset
topPosition
topWidget
traversalOn
traverseDown
traverseLeft
traverseRight
traverseUp
unmanageChild
unmapWidget
updateAccelerator
updateAcceleratorText
updateActivateCallback
updateArmCallback
updateBackgroundColor
updateBorderWidth
updateDisarmCallback
updateDragDetectCallback
updateForegroundColor

Managing Outboard Windows

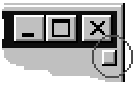
Collectively, all of the secondary windows that the WindowBuilder Pro editor controls are referred to as “Outboard” windows. If the **Update Outboards** option is turned on, WindowBuilder Pro will automatically update any outboard windows (e.g., attribute editors, attachment editor, etc.) with the currently selected widget or widgets.

All of the outboard windows close down automatically when the main WindowBuilder Pro window closes. If the “WindowBuilder Pro - Tools” configuration is loaded, you can close all the outboard windows at any time via the **Tools | Outboards | Close All** command.

Any of the outboard windows can become a floating window (e.g., one that floats above the main WindowBuilder Pro window as opposed to hiding behind it when the main window is selected).

To make an outboard window float:

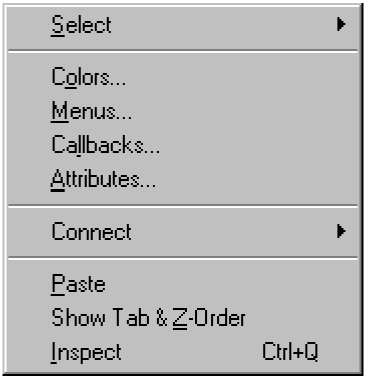
1. Select the outboard window that you would like to float.
2. Click the floating window button in the upper right corner of the window



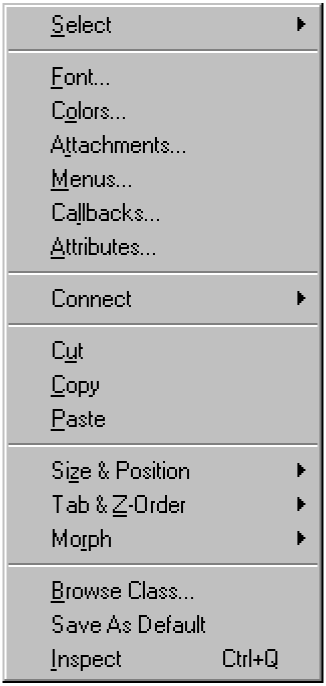
3. To remove this floating property from a window, click the floating window button again (it’s a toggle button).

Using Popup Widget Menus

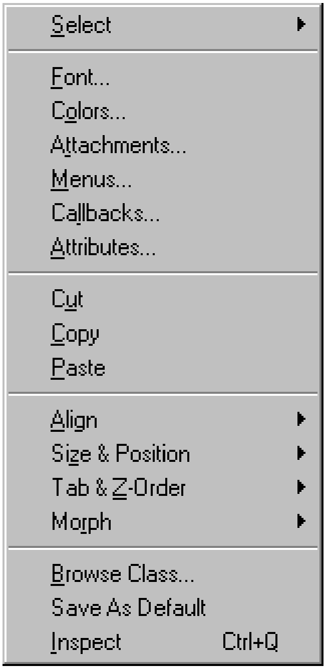
The WindowBuilder Pro editor provides popup context sensitive menus within the main editing window. If no widgets are selected, the window popup menu appears as shown below. This menu gives you quick access to setting the background color of the window, editing its menubar, callbacks (via the Callback Editor or visual connection) or attributes. It can also be used to select groups of widgets or to toggle the display of the tab & z-order.



If a single widget is selected, the widget popup menu appears as shown below. This menu gives you quick access to the widget's font, color, attachments, popup menu definition, callbacks and attributes. It also allows you to open a class browser on the class of the selected widget and gives you the option of saving the current widget as the default template for that type of widget. A cascading **Morph** menu provides a list of similar widget types that the current widget may be morphed to. The size, position and relative tab order may also be modified.



If multiple widgets are selected, the widget group popup menu appears as shown below. This menu gives you quick access to the widgets' font, color, attachments, popup menu definition, callbacks and attributes. It also allows you to open class browsers on the classes of the selected widget and gives you the option of saving the current widgets as the default templates for those types of widgets. A cascading **Morph** menu provides a list of similar widget types that the current widgets may be morphed to. Their size, position, alignment and relative tab order may also be modified.



If the optional "WindowBuilder Pro - Tools" configuration is loaded, each of the above menus also includes an Inspect option which allows the selected widget(s) to be inspected.

Chapter 5 Menus

WindowBuilder Pro supports hierarchical menus on both widgets and windows. When you attach a menu to a widget, it is called a popup menu. A menu under the title bar of a window is called a menubar. The process of adding menus to widgets and windows is almost the same. This section discusses the procedures that you use to create menubars for windows. Procedures specific to popup menu definition are discussed at the end of the section.

Creating a Menubar

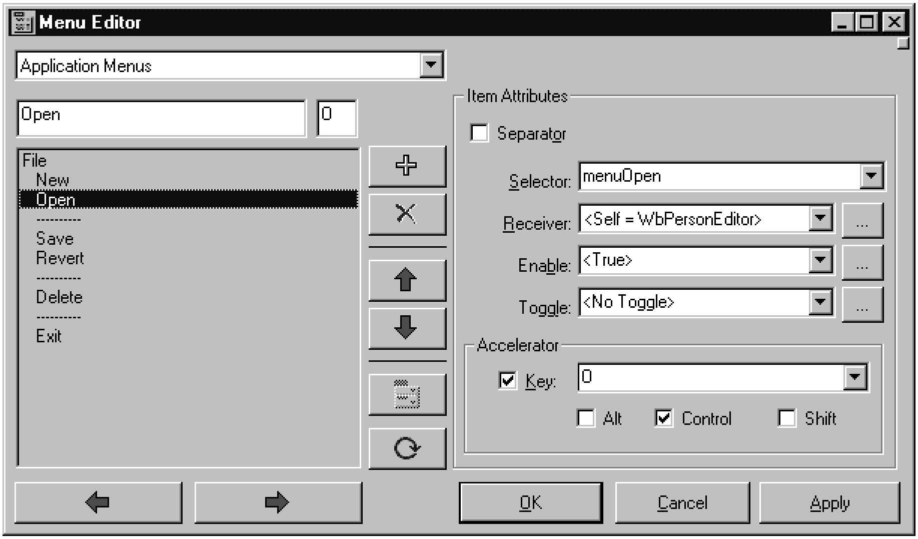
Menubars are useful for organizing functions in categories and making the categories available to the application user as menu titles. WindowBuilder Pro simplifies the menu building process by furnishing you with a menu editor.

To add a menu bar to a window:

- Select the title bar of the window, or a section of the window not covered by a widget.

Select the **Menus** command from the Attributes menu, or click the **Menu** button in the attribute bar.

The Menu Editor appears as shown below.



Application Menus

Open 0

File
New
Open

Save
Revert

Delete

Exit

Item Attributes

□ Separator

Selector: menuOpen

Receiver: <Self = WbPersonEditor>

Enable: <True>

Toggle: <No Toggle>

Accelerator

☑ Key: O

□ Alt ☑ Control □ Shift

If you have already created a menu bar, you can re-edit a particular menu by **double-clicking** it. This will open the Menu Editor with the appropriate menu group preselected.

To add a menu title to a menu:

1. Type the first menu title in the entry field. As you type, the menu title appears in the list box below the entry field.
2. Press Return when you are done. The menu title remains in the list box, and the entry field clears, ready for the next entry.

To add a menu item to a menu:

1. Click the ➡ button from the Menu Editor dialog. The selected menu shifts to the right.
2. Type the menu item. It will appear indented under its menu title.
3. Click the ⬅ button when you have entered all the menu items for that menu title. The selected menu shifts to the left. You are back at the menu title level, and you can enter another menu title.

Creating a submenu is the same process as the one just described, except you start a level lower.

To create a submenu:

1. Type the menu item you want to be the submenu title. Press Return to enter the menu item.
2. Click the ⇨ button, and type the first submenu item. Press Return.
3. Click the ⇦ button when you have entered all the submenu items for that submenu title. When you do so, the next item will be outdented to the next level up.

If your menu has many items, you can separate the items into different groups by using horizontal separator lines.

To insert a separator line:

1. Select the line where you want to put the separator line.
2. Click the ✚ button.
3. Click the Separator check box, or type a hyphen (“-”) in the entry field. A separator line appears in the list box.

Assigning Mnemonic Keys and Accelerator Keys

You can accommodate users who do not use a mouse by defining mnemonic keys and accelerator keys.

Mnemonic keys are combinations of ALT and some other key. These key combinations are assigned to menu titles and menu items. By typing the ALT-key combination, a user can open a menu, as if he had opened the menu with the mouse. The user can then access menu items by typing the letter assigned to it. Mnemonic keys display as underlined characters.

To assign a mnemonic key to a menu title:

1. Select the menu title to which you want to assign the mnemonic key.
2. Type an tilde “~” in front of the letter you want to be the mnemonic key, or type the desired character in the field next to the label field (the letter must be one of the characters in the label).

Keyboard accelerators are key combinations that execute menu items immediately when they are typed. They appear to the right of the menu object to which they are assigned.

To assign a keyboard accelerator to a menu item:

1. Select the menu item to which you want to assign the accelerator .
2. Select the **Key** text entry field in the **Accelerator** group box. Type the desired letter or select the desired key from the drop down list.
3. Check one or more of the **Alt**, **Control** or **Shift** key modifier boxes.

Editing a menu:

- To promote or demote an item after you have entered it, select the item, and click the ⇦ or ⇨ button.
- To move an item to another place in the menu, select the item and click the ⇧ or ⇩ button. Note that when you shift a submenu up or down, the submenu and all of its items will move.
- To insert a menu item, select the line where you want to insert the new item, and click the ✚ button. An empty space will appear. Enter the new item, and press Return.
- To delete a menu item, select the line that you want to delete, and click the ✕ button.

Each of these editing options can be applied to one or more menus at the same level in the hierarchy simultaneously. To select multiple menu items, hold the SHIFT or CTRL keys down while selecting the desired menu items. You can also drag select multiple menu items without holding down any modifier keys.

WindowBuilder Pro allows you to place menus in one of three groups to facilitate subclassing. These groups are Standard Left Menus, Application Menus and Standard Right Menus. You could, for example, define an abstract superclass that defines the File and Edit menus as the Standard Left Menus and the Help menu as the Standard Right Menus. Concrete subclasses would only need to implement menus in the Application Menus group.

To assign a menu to a group:

1. Select one of the top level menus (e.g., File, Edit, etc.).
2. Click the ▦ button. A dialog listing the other two groups will appear (its contents will vary based on what the current group is).
3. Select the group you want to move the menu to and hit **OK**.

To view menus in different groups:

- Select the desired group from the menu group combobox at the top left corner of the Menu Editor.

Most menu items have a selector associated with them. This is the message that will be sent to your application window when the menu item is selected.

To assign a selector to a menu item:

1. Select the menu item to which you want to add the selector.
2. In the **Item Attributes** group box, type the name of the selector in the **Selector** entry field. You can specify a unary menu message selector or a VisualAge callback-style 3 argument selector (just add a colon, WindowBuilder Pro will fill in the rest of the method name for you).WindowBuilder Pro will automatically generate matching skeletal methods for you.

Menu items may also be enabled, disabled, toggled on or toggled off dynamically.

To make a menu item enabled or toggled:

1. Select the menu item you wish to modify.
2. Select the desired enabled or toggled state.

and/or

3. Enter a method selector (in the same class as the receiver) that should (at runtime) answer true or false to specify the current state of the menu. You may also click the "..." button to the right of the fields to be presented with a list of unary messages understood by the receiver. By setting up these boolean method selectors, you can make menu management a snap at runtime.

Menu items may also be targeted at a receiver other than the application.

To change the receiver of the menu item:

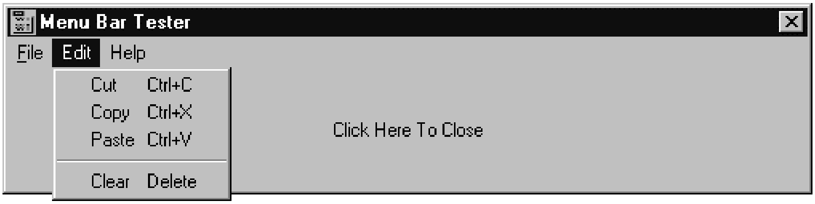
1. Select the menu item you wish to modify.
2. Select the receiver from the drop down list.

   Or, click the "..." button to get a list of classes that may be used as receivers (you could, for example, send the `#open` message to another WbApplication class)

When you have finished working on your menu, you can test it.

To test a menu:

- Choose ⟳. A new window will appear with a working example of your menu.



When you are satisfied with your menu, click **OK** to return to the main WindowBuilder Pro window. Click **Cancel** to return without saving the menu.

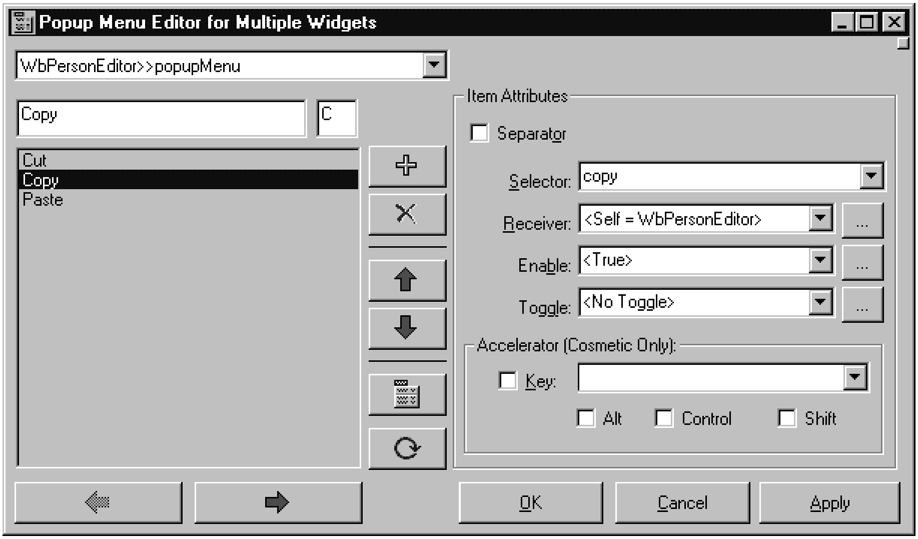
Popup menus on widgets

Most widgets can have popup menus associated with them. The process of creating a popup menu for a widget is the same as creating a menubar for a window. There are two minor differences:

- When you create a menubar, the top level menu items are menu titles that display horizontally on the menubar. When you create a popup menu, the top level menu items display vertically when the user right-clicks on the widget associated with the menu.
- When you test a menubar, a new window launches with a working example of your menu in it. When you create a popup menu, the menu pops up by itself, without its associated widget.

To add a popup menu to a widget

1. Select the widget or widgets that you wish to add a popup menu.
2. Select the **Menus** command from the Attributes menu, or click the **Menu** attribute bar button. The Popup Menu editor dialog appears as shown below.



Popup Menu Editor for Multiple Widgets

WbPersonEditor>>popupMenu

Copy | C

- Cut
- Copy
- Paste

Item Attributes

☐ Separator

Selector: copy

Receiver: <Self = WbPersonEditor>

Enable: <True>

Toggle: <No Toggle>

Accelerator (Cosmetic Only):

☐ Key:

☐ Alt ☐ Control ☐ Shift

The combobox that used to contain the menu category list in the Menu Bar editor, now contains a list of methods in your application that define popup menus. Initially this list will be empty except for "<None>." As new menus are defined, additional entries will appear in the list.

To create a new popup menu:

- Click the ▦ button. A prompter will appear in which you can enter the name of a method (zero arguments). Attempting to enter a menu item when <None> is selected, will automatically invoke this prompter and create a new menu.

Changing the popup menu assigned to a widget:

1. Select the desired popup menu from the combobox.
2. Click the **OK** or **Apply** button. Whichever popup menu is selected in the combobox will be assigned to the currently selected widget.

To remove a popup menu from a widget:

- Select "<None>" from the combobox.
- Click the **OK** or **Apply** button.

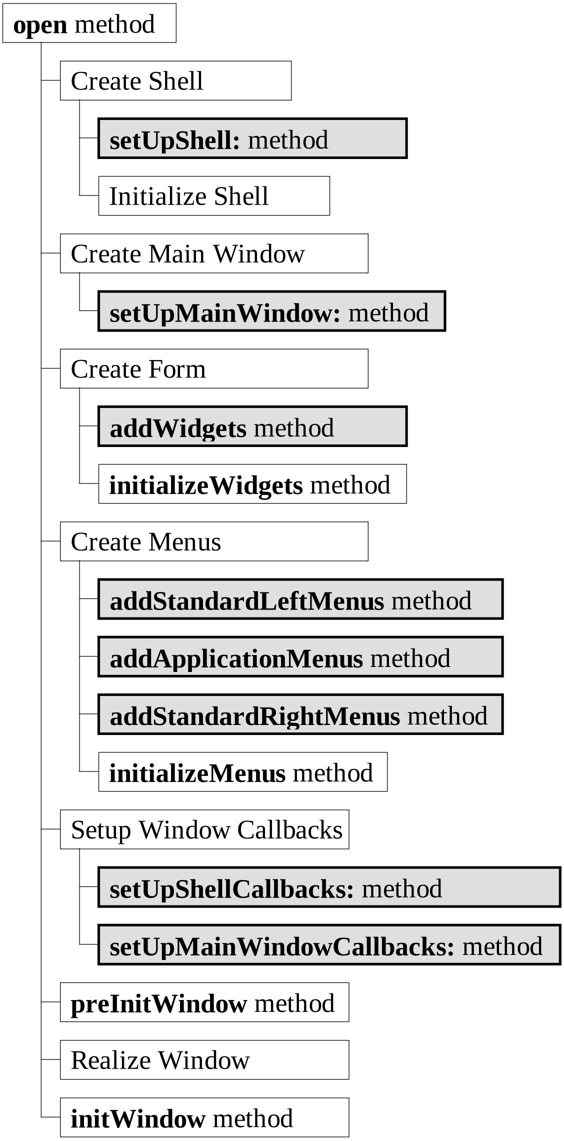
For more Information on the menu editor, see Chapter 8, *Command Reference*.

Chapter 6 Coding in WindowBuilder Pro

In this section, we will focus our attention on how you should create the pieces of your Smalltalk application for which WindowBuilder Pro doesn’t provide specific help. We’ll start by examining the code WindowBuilder Pro generates when you create a window or dialog. Then we’ll discuss how this code interacts with other elements of the Smalltalk system to create the user interface and framework for your applications. Finally, we’ll take a look at how you should approach this process to create user interface-related elements of your application that are outside the sphere of influence of WindowBuilder Pro.

WindowBuilder Pro and Smalltalk

When you launch your Smalltalk application after creating its interface in WindowBuilder Pro, the sequence of steps shown on the next page takes place. WindowBuilder Pro generated methods are shown in gray.


open method
Create Shell
setUpShell: method
Initialize Shell
Create Main Window
setUpMainWindow: method
Create Form
addWidgets method
initializeWidgets method
Create Menus
addStandardLeftMenus method
addApplicationMenus method
addStandardRightMenus method
initializeMenus method
Setup Window Callbacks
setUpShellCallbacks: method
setUpMainWindowCallbacks: method
preInitWindow method
Realize Window
initWindow method

Steps in Processing and Opening a Window

Smalltalk window based applications all have several important elements in common including the shell, the main window, the main form and a menubar if necessary. The `#open` method creates each of these elements in order. The code generated by WindowBuilder Pro has been factored into individual methods tasked with handling certain elements of this process.

The first element that is created is the shell. The `#setUpShell`: method that is generated by WindowBuilder Pro sets up attributes that are unique to the window as a whole such as its position, size, title and decorations (for example, frame style, minimize and maximize buttons, etc.). After the shell is created, the main window is created. The `#setUpMainWindow`: method sets up attributes such as the window's scrolling behavior.

Next the main form is created. This is the element to which the widgets are added. The `#addWidgets` method defines each of the widgets that make up the window as well as their attachments and callbacks. An optional hook method, `#initializeWidgets`, is provided to give you the opportunity to dynamically initialize any widget specific attributes that could not be captured by WindowBuilder Pro itself.

After the form, the menus are defined. WindowBuilder Pro splits the menu definitions into three groups: standard left menus, application menus and standard right menus. This allows an abstract superclass to define standard right and left menus (for example, File, Edit, Help) while subclasses only define unique application menus. Once the menus have been defined, an optional hook method, `#initializeMenus`, is provided. This provides an opportunity to add additional dynamically defined menus (or override any of the menus defined by WindowBuilder Pro).

After the menus have been defined, any callbacks for either the shell or the main window are defined in the `#setUpShellCallbacks`: and `#setUpMainWindowCallbacks`: methods.

Finally, there are two optional methods, `#preInitWindow` and `#initWindow`, that will be called automatically as part of the process of opening and displaying user interface elements in your Smalltalk application. The order in which these methods are called is important.

The `#preInitWindow` method is called before the window is realized and made visible. It provides a perfect place to add widgets and menus that WindowBuilder Pro can't handle but that must be defined before the window is made visible. It can also be used for such tasks as setting the contents of various panes with dynamically derived data that can't be hard-coded into WindowBuilder Pro because it isn't known until the program executes. The `#initWindow` method is called after the window is realized and made visible. It can be used in much the same way that the `#preInitWindow` method can be used. If there are any initialization tasks that require that the window be visible, this is the place to perform them.

The `#initializeWidgets`, `#initializeMenus`, `#preInitWindow` and `#initWindow` methods make it possible for you to do anything you want to a window or dialog without tampering with the methods generated automatically by WindowBuilder Pro.

What WindowBuilder Pro Generates

By default, WindowBuilder Pro generates a single method to define the widgets in your window or dialog. This method is called `#addWidgets`.

The basic structure for an `#addWidgets` method created by WindowBuilder Pro is something like this:

```
addWidgets
   create Widgets & define attributes
   define Attachments & Callbacks
```

The `#addWidgets` method is defined in two parts. The first half of the method creates the all of the widgets and defines all of their attributes. These attributes include things like the widget's upper left corner, size, labels, styles, contents, etc. The second half of the method defines attachments and callbacks for each widget.

Attachments define the widget's resizing behavior. You can specify all kinds of attachments. Attachments with respect to the application window, attachments with respect to any other widget in the form. Proportional attachments and fixed size attachments can also be specified.

Callbacks are a mechanism by which the application is notified when some higher level action is performed on a widget. For example, the Activate Callback is used to inform the application that a CwPushButton has been pressed and released. The Destroy Callback is used to inform the application that a widget has been destroyed. The application can take some appropriate action via the callback method in response to this action.

The callback method is a 3 argument method such as `#browseClass:clientData:callData`. The first argument, "aWidget", is the widget which caused this callback to occur. The second argument, "clientData", is any application-specific data you might choose to send. The last argument, "callData", is a widget-specific object whose contents depend upon the widget and the nature of the callback. When you add a callback to a widget, WindowBuilder Pro automatically generates the corresponding stub method in your application.

Notice the comment at the beginning of the method. It includes a warning that it is not particularly wise to change this method. This is because the next time you edit this window or dialog and save it, WindowBuilder Pro generates a new `#addWidgets` method, overwriting the existing one, if any. Later in this chapter, we'll see how to get around this necessary limitation.

Passing Arguments to Windows

When a window is opened, it often launches with some initial information already filled in. For example, a message box may have a string of text to display, a color dialog may start with a currently selected color, or a font dialog may start with a currently selected font. As a designer, you will probably want to create windows of your own with similar functionality.

Imagine a simple window called “ExamplePrompter”. It requires two pieces of information to start up: the text used to prompt the user, and the initial text placed in the text field. To pass this information in, we might want to launch the window with the following syntax:

```
ExamplePrompter new
   prompt: ‘Enter a new exclamation:’
   default: ‘Aaaargh!’.
```

This requires that we create an instance method in ExamplePrompter called `#prompt: default`:. This method must take in these two arguments, open the window, and set the values of the static label and text field. Let’s see how that can be done.

We’ll start with the `#prompt:default`: method:

```
prompt: string1 default: string2
   promptString := string1.
   responseText := string2.
   self open.
```

In this method, we store the two strings passed in using instance variables we’ve declared, then execute the open method which in turn calls the `#addWidgets` method generated by WindowBuilder Pro. Later, during the initialization process that occurs during the open method, we’ll make use of these instance variables to set the contents of the various controls:

```
initWindow
   (self widgetNamed: ‘promptText’) labelString: promptString.
   (self widgetNamed: ‘editor’) value: responseText.
```

That’s all there is to it! As you can see, it’s really very easy to make use of arguments in code without altering any WindowBuilder Pro generated methods.

Returning Values From a Dialog

So far, we’ve modified the ExamplePrompter dialog so that it accepts arguments when it is initialized. How do we actually make use of the information the user enters? For that matter, how do we even close the window?

Let’s deal first with the process of dismissing the dialog. When the user presses the OK button, they expect the window to close. Let’s see to it that this happens. WindowBuilder Pro generates an `#ok:clientData:callData`: method for us when we tell it to use that method as the response to a user click on the OK button. Modify the empty `ok:` method to look like this:

```
ok: aWidget clientData: clientData callData: callData
   self close.
```

Now the window can be closed, but a big issue remains: the method which invoked this prompter wants some information from the user—that’s why it launched the dialog in the first place. The question is, how can our dialog offer this information once the user has filled it in?

The easiest way to do so is to query the dialog after it returns. Since this is a dialog, the `#prompt:default`: method will not return until the window is closed. All we need to do is store the necessary information in instance variables after the dialog is dismissed, and provide accessor methods to these instance variables. Then we can simply use these accessor methods to ask the dialog for the information.

For example, if we add a method result that answers the user’s response, we can then use the following code:

```
exclamation:=
   (ExamplePrompter new
      prompt: ‘Enter a new exclamation:’
      default: ‘Ddoooooooh!’) result.
```

The result method is straightforward: we can use the instance variable responseText again, like so:

```
result
   ^responseText
```

But there’s a problem here. This will always return the initial value of the responseText, since it’s never set to the contents of the text field. Let’s take care of this. Alter the `#ok:clientData:callData`: method as follows:

```
ok: aWidget clientData: clientData callData: callData
   responseText := self widgetNamed: ‘editor’) value.
   self close.
```

This will ensure that the instance variable is set up correctly for the result method.

The only issue that remains is the Cancel button. This typically means the user has decided to cancel the change they were going to make; we need some way of communicating this back from the dialog. A commonly accepted convention under such circumstances is to return nil, and this is easy to do. We simply have the cancel: method set the responseText instance variable to nil before closing the window, as follows.

```
cancel: aWidget clientData: clientData callData: callData
   responseText := nil.
   self close.
```

With that, we’ve completed the interactive portions of the ExamplePrompter. The techniques used here are only one way of accomplishing the tasks at hand, but provide a general mechanism that works under many different circumstances.

Naming Widgets

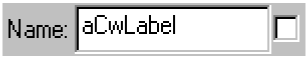
As we saw in the previous section, sending messages to individual widgets is easy. Each widget defined in a window must have a name. By default, WindowBuilder Pro generates a name of the form “a<WidgetClass>n” where “n” is a number. In general, you will want to give your widgets more descriptive names such as “okButton”, “addressLabel”, “nameField”, etc. Sending a message to named widget is simple. To ask the application for a particular widget, use the `#widgetNamed`: protocol like this:

```
(self widgetNamed: ‘nameField’)
```

To set the contents of a text field, therefore, you would do the following:

```
(self widgetNamed: ‘nameField’) value: aNameString
```

WindowBuilder Pro also gives you the option of directly assigning any widget to be an instance variable of the application (via the checkbox next to the widget name).



Name: aCwLabel □

If you do this, the above expression can be re-written as:

```
nameField value: aNameString
```

where “nameField” is now an instance variable.

Passing messages from one widget to another

Now that you know how to address a widget programatically and send it messages, sending messages from one widget to another is easy. As an example, let’s assume that we have two widgets, a single select WbScrolledList and a CwLabel, and that we want the selected item of the list to update the contents of the label. This code can be expressed as:

```
(self widgetNamed: ‘aCwLabel’) labelString:
   (self widgetNamed: ‘aWbScrolledList’) selectedItems first
```

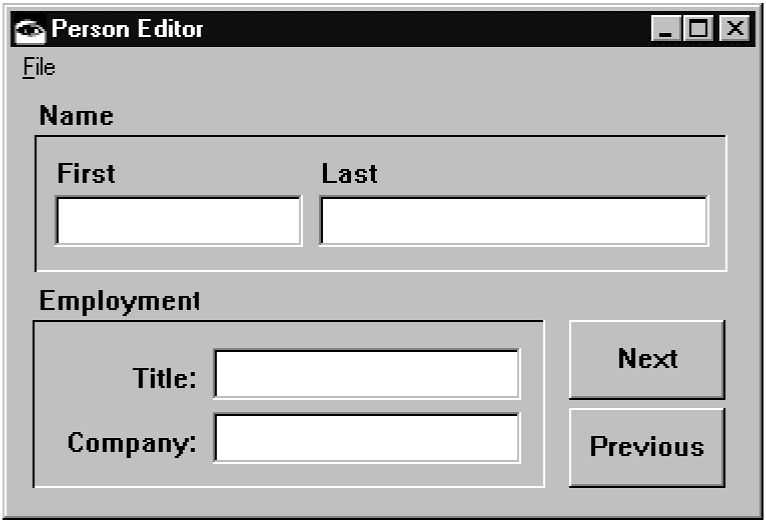
or more directly using instance variable assigned widgets as:

```
aCwLabel labelString: aWbScrolledList selectedItems first
```

Chapter 7 Example Application

To give you some hands-on experience using WindowBuilder Pro, this chapter presents a simple data entry application. The Person Editor application tracks data that you input, and you are able to edit and add to the data. It is a very simple application but does a good job of highlighting the important steps in building a window. All of the windows in WindowBuilder Pro, including WindowBuilder Pro itself, were built using the same techniques. If you are interested in further, more sophisticated examples, examine the windows that make up WindowBuilder Pro (source code is provided for many of them in the Team version).

The finished Person Editor application looks like this.



The finished Person Editor application.

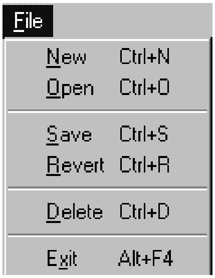
This section presents the steps to follow to create the sample application. The application is intended to give you experience with the procedures that are covered in previous chapters of the manual. Refer to those chapters for more information on these procedures. If you perform the steps as instructed, you will touch on the important features of WindowBuilder Pro. In keeping with the format of the rest of the manual, the steps are divided into two sections: Designing the Interface, and Attaching Callbacks and Event Handlers to Widgets. You probably would not actually create an application in this manner; more likely, you would attach callbacks and event handlers as you think of them.

In the steps that follow, positions and sizes of widgets are given so that you may duplicate the application exactly as it is shown on the prior page. If duplicating the exact positions of the widgets is not important to you, you can just place the widgets by eye. Since a working version of the Person Editor application is already provided with WindowBuilder Pro (it is the class WbPersonEditor in the WbProTutorialExample application), you can display the Person Editor in WindowBuilder Pro and follow along with the text.

Be sure to save the window occasionally as you work. The first time you do, you will be prompted to save your window as a subclass of WbApplication. Since you also need to provide an application into which the class can be saved, you may want to create a new application with the Application Manager before you start.

Designing the Interface

1. From the File menu, choose **New**.
2. Resize the window to 372 x 212 by clicking the Size button on the Status bar. Double-click anywhere in the window to bring up a window editor, and type **Person Editor** in the title field. Alternatively, you can simply **ALT-click** on the titlebar to direct edit the title of the window.
3. From the Attributes menu, choose **Menus**, or click the **Menu** button to open the Menu editor. Build the menu shown below. Recall that mnemonic keys are displayed as underlined characters in menu items, and accelerator keys are displayed on the right of the menu. Using the picture as a guide, add accelerator and mnemonic keys to the menu.



Person Editor Application Menu.

4. Place a CwText widget at position 20,52 with a size of 126x28. Name the widget firstName by typing **firstName** in the name field on the attribute bar.

5. Place a CwText widget at position 152,52, with a size of 197x28. Name the CwText widget **lastName**.

6. Place a CwLabel widget at position 20,32. Name the widget **firstNameLabel**. Open an CwLabel editor by double-clicking on the widget. In the Label String field, type **First**.

7. Place a CwLabel widget at position 152,32. Name the label **lastNameLabel**. Open the CwLabel editor and Type **Last** in the Label String field. Click OK to close the editor. As an alternative to using the attribute editors, you can **ALT-click** on the widgets to direct edit their labels.

8. Place the first of two CwText widgets in the lower part of the window, by using the right mouse button (this leaves the cursor loaded with the widget). Then place the second CwText widget below the first, using the left mouse button (which unloads the cursor). Select the top widget, and set its position to 100, 128, and its size to 156x28.

9. Leaving the upper widget selected, **SHIFT-select** the lower widget. From the Size menu, choose **Replicate Height**, and then **Replicate Width**, (or click the button equivalents). The two widgets are now the same size. Position the lower widget at 100, 160. Click a blank part of the window to deselect the widgets.

10. Select the upper widget, and name it **title**. Select the lower widget, and name it **company**.

11. Place a CwLabel widget at position 20, 132. Name the label **titleLabel**. Open a CwLabel editor and type **Title:** in the **Label String** field. In the Alignment box, select Right. Uncheck Recompute Size. Click **OK** to close the editor. Keep the widget selected.

12. Copy the titleLabel widget by choosing Copy from the Edit menu. Paste the copy at position 20,164. Name the new label **companyLabel**. Open aCwLabel editor and type **Company:** in the Label String field. Note that the attribute settings are already correct, since you copied them from the *titleLabel* widget. Click **OK** to close the editor.

13. Select a WbFrame, and drag it so that it surrounds the *firstName* and *lastName* widgets and their labels. The position of the frame in the example shown above is 12,28, and its size is 346x68. Name the frame **nameFrame**. Open a WbFrame editor. In the **Shadow Type** box, select In. In the **Frame Thickness** field, type **1**. Click **OK** to close the editor.

14. Make a copy of the WbFrame *nameFrame*. Paste and drag the copy so that it surrounds the title and company widgets and their labels. The position of the frame in the example shown above is 12,116, and its size is 260x84. Name the frame **employmentFrame**.

15. Place a CwPushButton at position 280, 116, and set the size to 78x40. Name the button **next**. Open a CwPushButton editor. In the **Label String** field, type **Next**. Uncheck the Recompute Size box. Click **OK** to close the editor.
16. Place a CwPushButton at position 279,160. Name the button **previous**. Select the "next" button, then **SHIFT-select** the "previous" button. From the Size menu, choose **Replicate Height**, then **Replicate Width**. The buttons should now be the same size. Click elsewhere in the window to unselect the widgets. Select the "previous" button. Open a CwPushButton editor. In the **Label String** field, type **Previous**. Uncheck the Recompute Size box. Click **OK** to close the editor. As stated earlier, you always have the option of **ALT-clicking** on a widget to direct edit its contents.
17. Save the application.

You have completed the user interface part of the application. If you want, you can test the window by choosing **Test Window** from the File menu. The widgets and menu should respond. Nothing else happens, because there is no code attached to the widgets. In the next section, you attach callbacks and event handlers to the widgets to give the application some behavior.

Attaching Callbacks and Event Handlers

1. Open the Menu editor. Select the New menu item. Notice that the Selector box is enabled. For each menu item in the chart, type the corresponding method in the Selector box.

Menu Item	Method
New	menuNew
Open	menuOpen
Save	menuSave
Revert	menuRevert
Delete	menuDelete
Exit	menuExit

2. Select the firstName CwText widget. Open a callback editor by choosing Callbacks... from the Attributes menu, or clicking the equivalent button. From the Callback List, select Value Changed Callback. In the Method combobox, type "textChanged::", and press Return. Be sure to include the two colons. Notice that WindowBuilder Pro automatically appended the standard second and third key words (clientData:callData:) to the method name. Click OK to close the editor.
3. Using the chart below, attach callbacks for the remaining widgets by repeating the procedures in Step 2. The "firstName" callback is included for reference. Note that once the Callback Editor is open, there is no need to close it until you are done attaching callbacks. You may either use the combobox at the top of the screen to

select another widget or click on the widget in the main editing window (the Callback Editor and any other open editors will automatically update to reflect the currently selected widget).

Widget Name	Callback List Item	Method Prefix
firstName	Value Changed Callback	textChanged
lastName	Value Changed Callback	textChanged
title	Value Changed Callback	textChanged
company	Value Changed Callback	textChanged
next	Activate Callback	next
previous	Activate Callback	previous

4. Save the application. WindowBuilder Pro will regenerate the window's layout and menu definitions (e.g., the `#addWidgets` and `#addApplicationMenus` methods) and create method stubs for any callbacks and menus that you specified.

You have completed attaching callbacks to the widgets. The next step is to add program logic to the generated stub methods. For the Person Editor application, the necessary methods are included in the WbPersonEditor class. You can either type them into your class definition or copy the ones that have been provided. So you may now test your application.

You now have a working application with buttons, text fields, and a menu. Feel free to modify and expand the application. Some of the enhancements that you might consider are defining attachments (resize behavior) for each of the widgets as well as an overall tab order.

Chapter 8 Command Reference

This chapter provides a detailed description of each command available within WindowBuilder Pro. The first two pages establish a name for each toolbar function. Each of these function is described in detail in the context of its corresponding menu command. All of the WindowBuilder Pro menus (both on the Transcript and within the main editor) are discussed immediately following the identification of the toolbar buttons.

Main Toolbar

Test
New
Open
Save
Cut
Copy
Paste
Duplicate
Delete
Undo
Redo
Morph
Bring To Front
Send To Back

Auto Size
Replicate Width
Replicate Height
Align Left
Align Center
Align Right
Align Top
Align Middle
Align Bottom
Distribute Horizontally
Distribute Vertically
Templates
Properties

Attribute Toolbar

Font

Color

Attachments

Menus

Callbacks

Attributes

Status Bar

Select All

Select All In Class

Browse Class

Tab & Z-Order

Set Position

Set Size

Show Grid

Widget Categories

Text

Buttons

Lists

Composites

Sliders

Notebooks

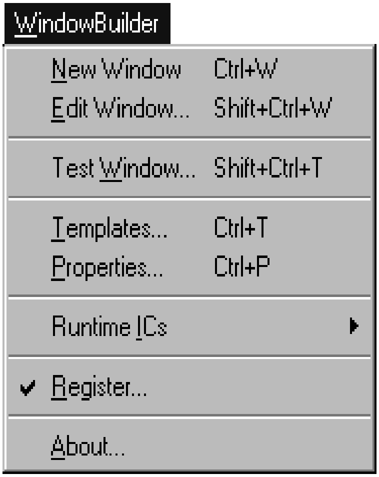
Containers

Other/Miscellaneous

Windows 95

OLE/ActiveX

Transcript



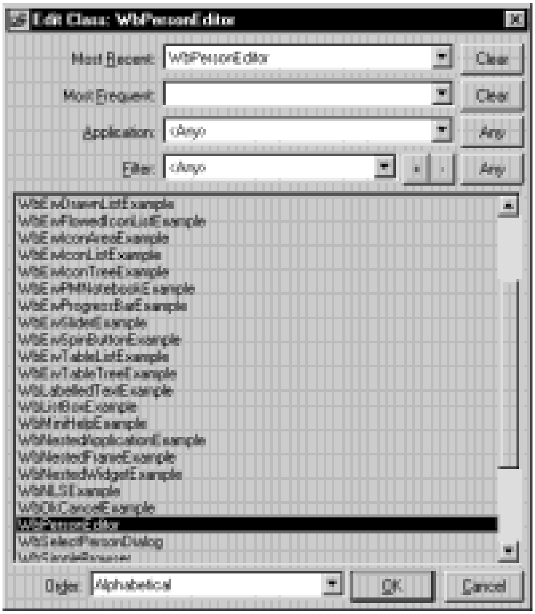
New Window

Creates a new window. WindowBuilder Pro starts from scratch on a new copy of the default template window.

Edit Window...

Edit an existing window definition (WbApplication subclass). WindowBuilder Pro will prompt the user with an Edit Class dialog containing a list of windows that have been created by it. When a window is chosen, WindowBuilder Pro will begin editing the window of the selected WbApplication subclass.

By default, the list in this dialog contains only those windows built by WindowBuilder Pro. The classes are listed alphabetically. They can also be viewed hierarchically or by frequency of access. A handy application filter is provided to help you look at just the classes that you wish. As a convenience, WindowBuilder Pro keeps track of the most recently accessed and most frequently accessed classes.


Edit Class: WbPersonEditor

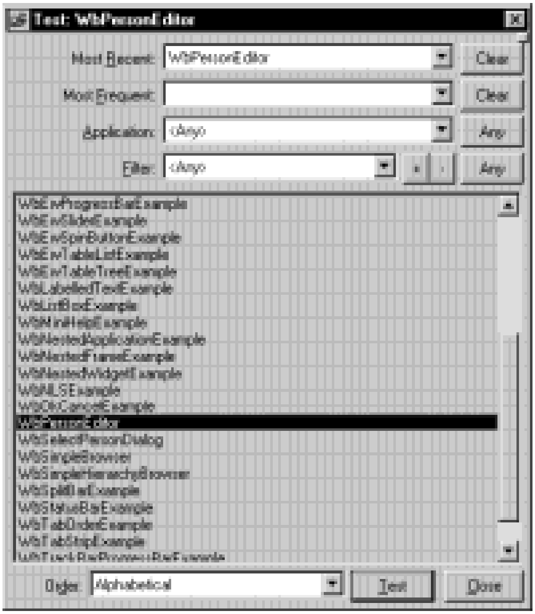
Most Recent: WbPersonEditor
Most Frequent:
Application: <Any>
Filter: <Any>

WbEwDrawnListExample
WbEwFlowedIconListExample
WbEwIconAreaExample
WbEwIconListExample
WbEwIconTreeExample
WbEwPMNotebookExample
WbEwProgressBarExample
WbEwSliderExample
WbEwSpinButtonExample
WbEwTableListExample
WbEwTableTreeExample
WbLabelledTextExample
WbListBoxExample
WbMiniHelpExample
WbNestedApplicationExample
WbNestedFrameExample
WbNestedWidgetExample
WbNLSExample
WbOkCancelExample
WbPersonEditor
WbSelectPersonDialog
WbSimpleBrowser

Order: Alphabetical

Test Window...

Test an existing window definition (WbApplication subclass). WindowBuilder Pro will prompt the user with an Test dialog containing a list of windows that have been created by it. When a window is chosen, an instance of that window will be created and opened.


Test: WbPersonEditor

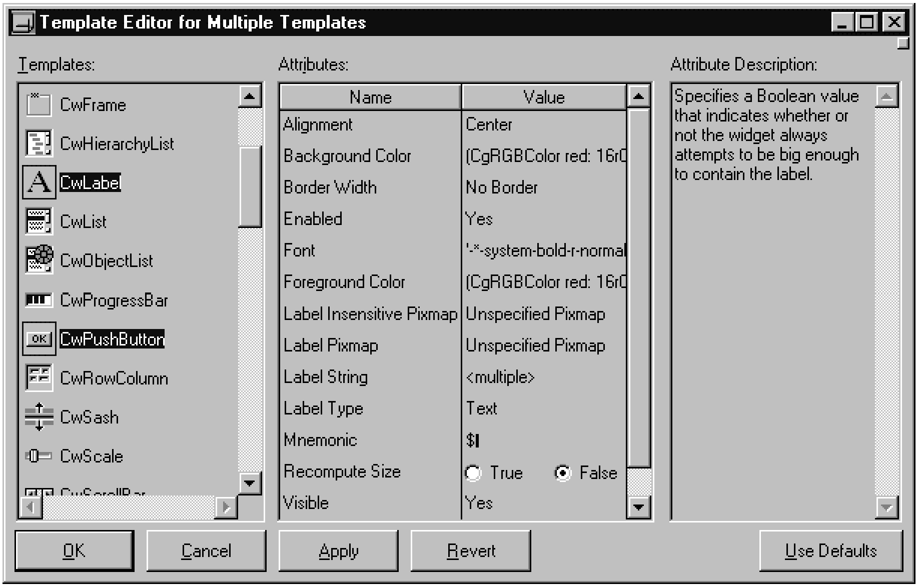
Most Recent: WbPersonEditor
Most Frequent:
Application: <Any>
Filter: <Any>

WbEwProgressBarExample
WbEwSliderExample
WbEwSpinButtonExample
WbEwTableListExample
WbEwTableTreeExample
WbLabelledTextExample
WbListBoxExample
WbMiniHelpExample
WbNestedApplicationExample
WbNestedFrameExample
WbNestedWidgetExample
WbNLSExample
WbOkCancelExample
WbPersonEditor
WbSelectPersonDialog
WbSimpleBrowser
WbSimpleHierarchyBrowser
WbSplitBarExample
WbStatusBarExample
WbTabOrderExample
WbTabStripExample
WbTrackBarProgressBarExample

Order: Alphabetical

Templates...

Displays the template editor.



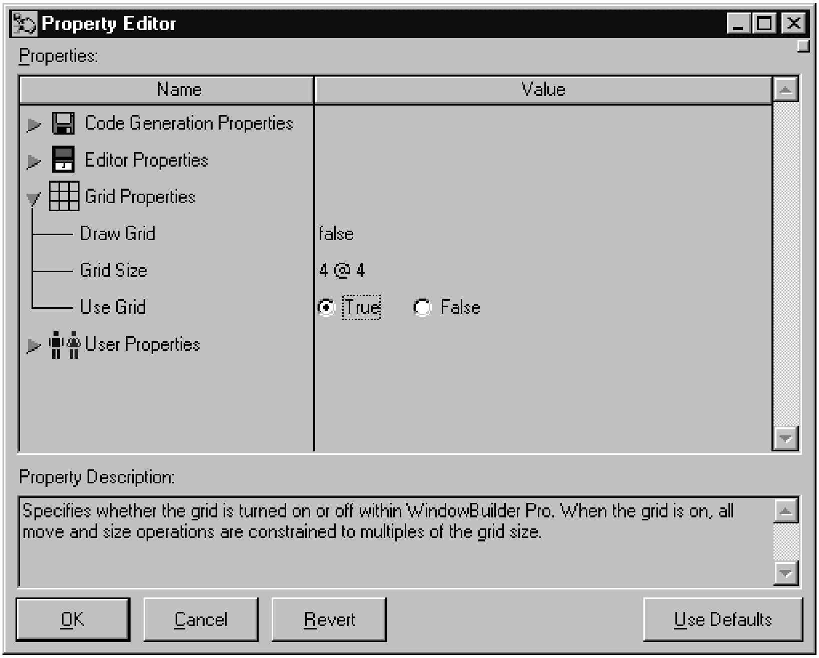
Name	Value
Alignment	Center
Background Color	(CgRGBColor red: 16r0
Border Width	No Border
Enabled	Yes
Font	'-*-system-bold-r-normal
Foreground Color	(CgRGBColor red: 16r0
Label Insensitive Pixmap	Unspecified Pixmap
Label Pixmap	Unspecified Pixmap
Label String	<multiple>
Label Type	Text
Mnemonic	$I
Recompute Size	○ True ● False
Visible	Yes

Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

The template editor allows you to specify default attribute values for attributes of all the widgets supported by WindowBuilder Pro. These widgets are the concrete classes in the Common Widgets and the Extended Widgets subsystem of VisualAge. Once you change a default value, that new value will be used for all new instances of that widget that you place in the editor. For example, all CwArrowButtons are upward pointing by default. This can easily be changed so that all new buttons are right pointing when first dropped into the editor.

Properties...

Displays the property editor.



Code Generation Properties
Editor Properties
Grid Properties
Draw Grid false
Grid Size 4 @ 4
Use Grid ● True ○ False
User Properties

Property Description:

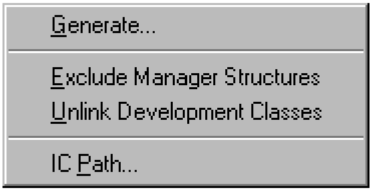
Specifies whether the grid is turned on or off within WindowBuilder Pro. When the grid is on, all move and size operations are constrained to multiples of the grid size.

The properties editor is used to customize your WindowBuilder Pro environment. You can customize the code generation properties, the editor properties, grid properties and the user properties.

See the discussion of the Property Editor in the section on the Option menu for a list of all available properties.

Runtime ICs...

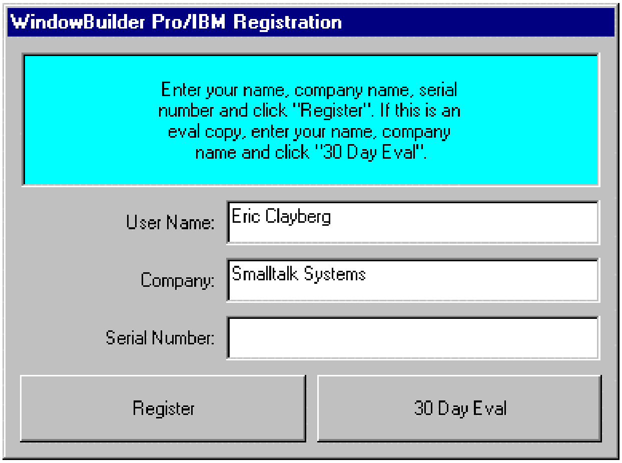
Generate ICs for all runtime components.



The first option is to generate the ICs. This is a long running process that can be cancelled while in progress. The second option specifies whether ENVY manager structures should be excluded. This is the information that specifies versions, method categories and class ownership. An IC generated with this option turned on will only be loadable in a runtime image. The next option specifies whether instances of development classes (e.g., EmTimeStamp) should be unlinked during export. An IC generated with this option turned off will not be loadable at runtime. For a runtime loadable IC, both of these options should be turned on; for a development loadable IC, they should be turned off. See the IBM Smalltalk User's Guide, pp 314-6 for a more complete description of these last two options. The final option specifies the directory in which to generate the ICs. The default is the current directory (e.g., ‘.’)..

Register...

Displays the registration dialog.



WindowBuilder Pro/IBM Registration

Enter your name, company name, serial number and click "Register". If this is an eval copy, enter your name, company name and click "30 Day Eval".

User Name: Eric Clayberg

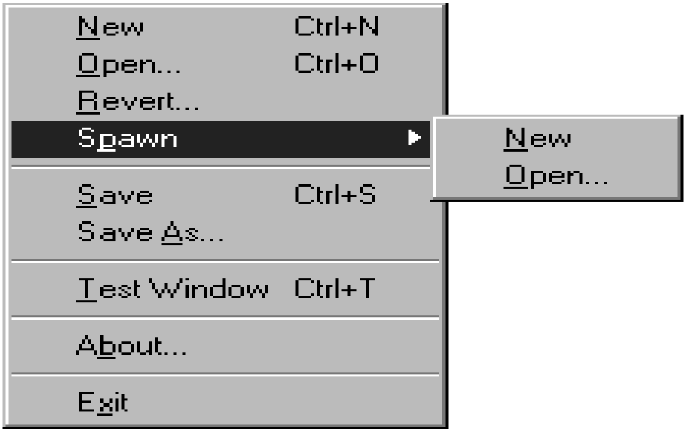
Company: Smalltalk Systems

Serial Number:

About...

Displays information about WindowBuilder Pro.

File

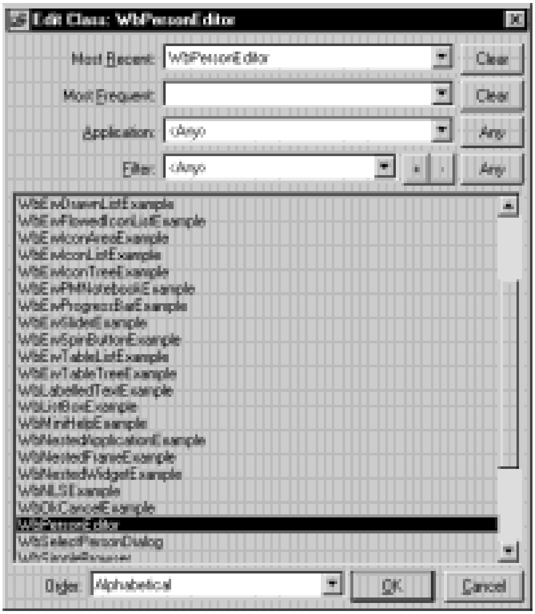


New

Creates a new window. WindowBuilder Pro discards any work in progress, and starts from scratch on a new copy of the default template window.

Open...

Edit an existing window definition (WbApplication subclass). WindowBuilder Pro will prompt the user with an Edit Class dialog containing a list of windows that have been created by it. When a window is chosen, WindowBuilder Pro will discard any work in progress, and begin editing the window of the selected WbApplication subclass.



Edit Class: WbPersonEditor

Most Recent: WbPersonEditor

Most Frequent:

Application: <Any>

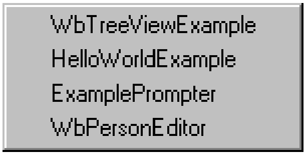
Filter: <Any>

WbEwDrawnListExample
WbEwFlowedIconListExample
WbEwIconAreaExample
WbEwIconListExample
WbEwIconTreeExample
WbEwPMNotebookExample
WbEwProgressBarExample
WbEwSliderExample
WbEwSpinButtonExample
WbEwTableListExample
WbEwTableTreeExample
WbLabelledTextExample
WbListBoxExample
WbMiniHelpExample
WbNestedApplicationExample
WbNestedFrameExample
WbNestedWidgetExample
WbNLSExample
WbOkCancelExample
WbPersonEditor
WbSelectPersonDialog

Order: Alphabetical

By default, the list in this dialog contains only those windows built by WindowBuilder Pro. The classes are listed alphabetically. They can also be viewed hierarchically or by frequency of access. A handy application filter is provided to help you look at just the classes that you wish. As a convenience, WindowBuilder Pro keeps track of the most recently accessed and most frequently accessed classes.

Right-clicking on the **Open** button in the toolbar will popup a menu listing the most recently edited windows.



WbTreeViewExample
HelloWorldExample
ExamplePrompter
WbPersonEditor

Spawn

Displays a submenu with the following choices:

New

Opens a new copy of WindowBuilder Pro on the currently edited window.

Open

Opens a new copy of WindowBuilder Pro on a window selected from the Edit Class dialog.

Revert...

Revert the last saved version of the window. Any changes that have not been saved will be lost.

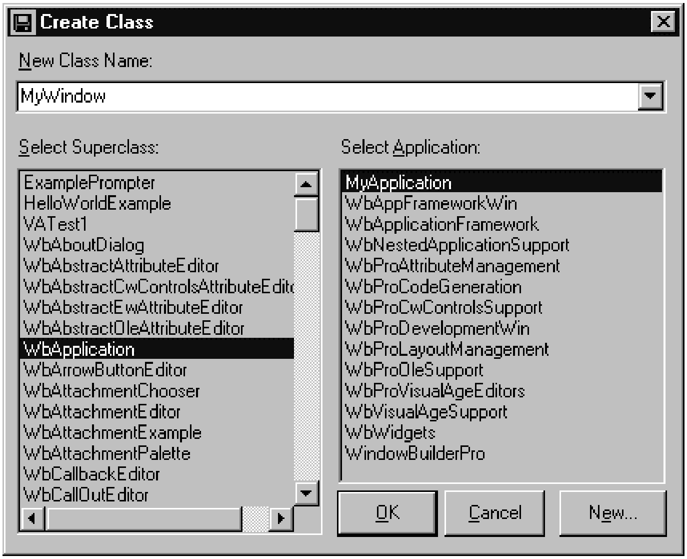
Save

Save the edited window by generating the appropriate code. If the edited window has not been associated with a class, this command behaves exactly as the Save As command, described below.

Warning: When a window’s definition is saved, it writes over the previous methods for the window’s layout and menu structure. You can use the standard ENVY tools to load previous versions if need be.

Save As...

Save the edited window by generating the appropriate code. When this command is executed, WindowBuilder Pro will bring up the following dialog from which to select a target application window:



Create Class

New Class Name: MyWindow

Select Superclass: WbApplication

Select Application: MyApplication

In this dialog you select a superclass, you specify a class and specify an application in which you want the class to be created. You must have an open edition of a application already created using the Smalltalk Application Manager. Otherwise you will not be able to save the window definition. If this happens, cancel out of the Create Class dialog. Go to the Application Manager and create a new application or create a new edition of an existing application. You can also click the New button to automatically create a new application with the selected superclass as a prerequisite.

Remember the application into which you are trying to create the window class must have WbApplicationFramework in its prerequisite chain. The classes in application WbApplicationFramework constitute the runtime portion of WindowBuilder Pro.

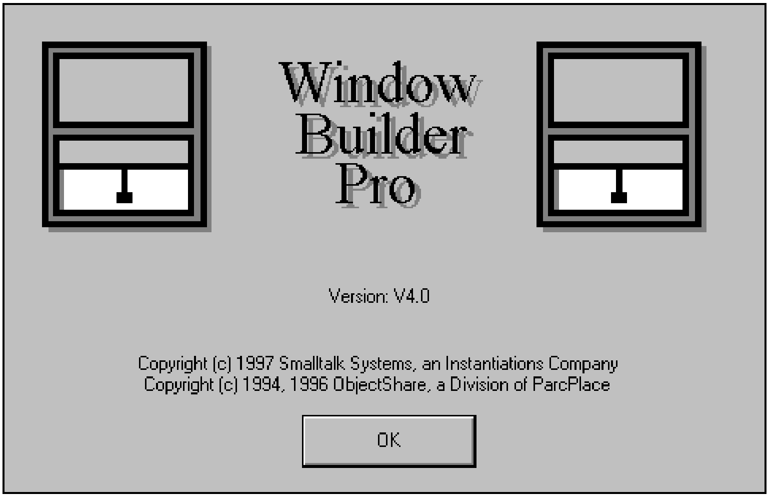
Warning: When a window’s definition is saved, it writes over the previous methods for the window’s layout and menu structure. You can use the standard ENVY tools to load previous versions if need be.

Test Window

Launches an example of the currently edited window, by sending the message open to a new instance of it. This function can also be performed by pressing the Test button.

About...

Displays information about WindowBuilder Pro.



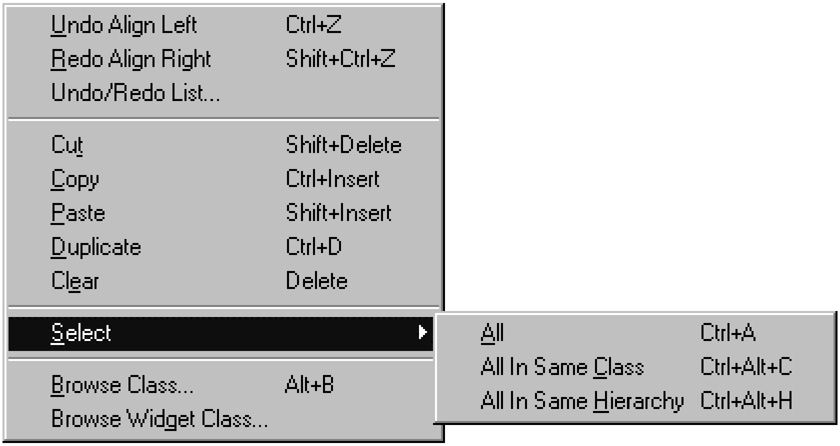
Version: V4.0

Copyright (c) 1997 Smalltalk Systems, an Instantiations Company
Copyright (c) 1994, 1996 ObjectShare, a Division of ParcPlace

Exit

Exit WindowBuilder Pro. If any changes have been made since the last save of the application window, the user will be prompted to save the window. Selecting Close from the System menu has the same effect.

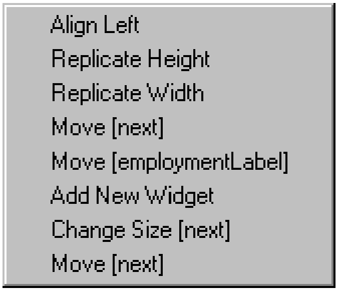
Edit



Undo

Undoes the last edit operation. This command is dimmed when an operation cannot be undone. The number of available undo levels is configurable from the WindowBuilder Pro Property editor.

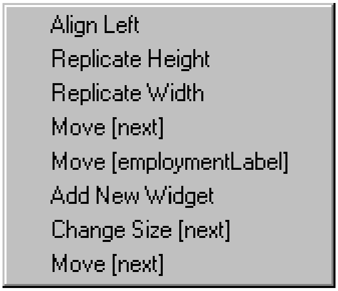
Right-clicking on the Undo toolbar button pops up a menu containing all of the edit operations that can be undone.



Redo

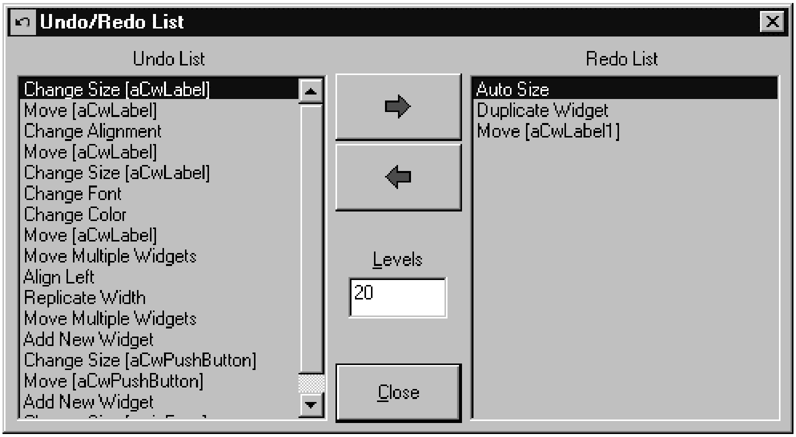
Redoes the last edit operation that was previously undone. This command is dimmed when an operation cannot be redone.

Right-clicking on the **Redo** toolbar button pops up a menu containing all of the edit operations that can be redone.



Undo/Redo List...

Launches a dialog from which multiple edit operations can be undone or redone. The maximum number of undo levels may also be set.



Levels: 20

Cut

Removes the selected widget(s) from the design surface, placing it (them) on the clipboard for later pasting.

Copy

Places a copy of the selected widget(s) on the clipboard.

Paste

Loads the cursor with the widgets on the clipboard. When the mouse is clicked somewhere within the design surface, the clipboard selection is placed at that location.

Duplicate

Creates another copy of the selected widget or widgets. The first duplicate will be diagonally offset from the original. If this copy is repositioned, subsequent duplicates will be placed using this new offset.

Clear

Removes the selected widget(s) from the interface without affecting the clipboard. This is the same as pressing the Delete key on the keyboard.

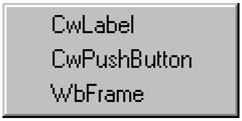
Select

Displays a submenu with the following choices:

Select All

Selects all the widgets within the main window.

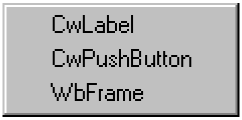
Right-clicking on the **Select All** button in the lower left corner of the WindowBuilder Pro window will popup a menu listing all of the widget types used in the layout. Selecting one will cause all widgets of that type to be selected.



Select All In Same Class

Selects all the widgets in the same class. This command can be used as a fast path for selecting all labels or buttons on the screen.

Right-clicking on the **Select All In Same Class** button in the lower left corner of the WindowBuilder Pro window will popup a menu listing all of the widget types used in the layout. Selecting one will cause all widgets of that type to be selected.



Select All In Same Hierarchy

Displays all the widgets in the same hierarchy.

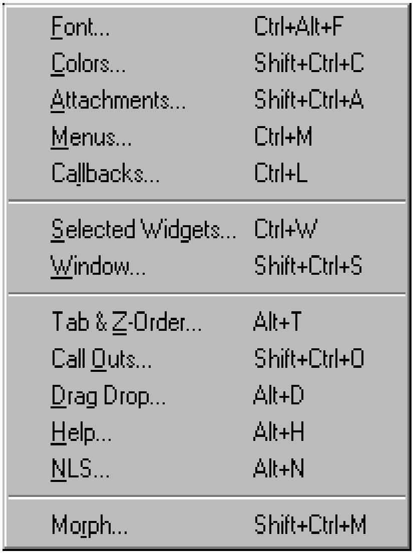
Browse Class...

Display a standard class browser on the currently edited WbApplication subclass. If the class has not been saved yet, you will be prompted to save the class.

Browse Widget Class...

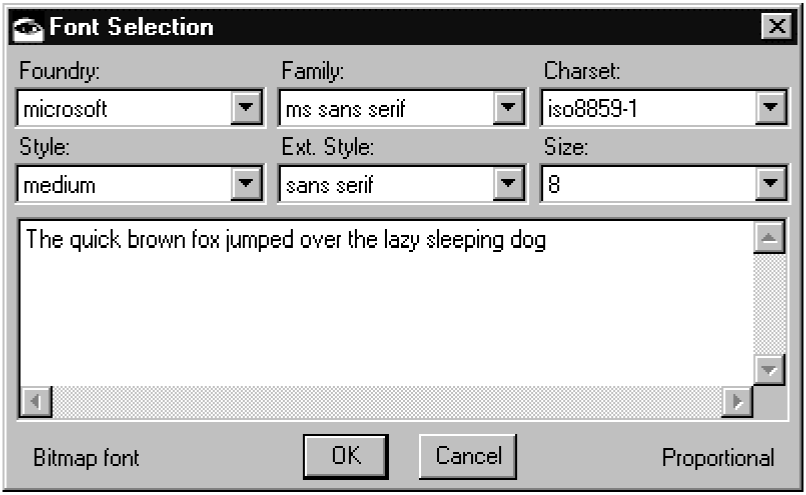
Display a standard class browser on the class of the currently selected widget.

Attributes



Font...

Displays a Font Selection dialog for the currently selected window or widget. This is the standard font dialog provided with VisualAge.



Foundry: microsoft
Family: ms sans serif
Charset: iso8859-1
Style: medium
Ext. Style: sans serif
Size: 8

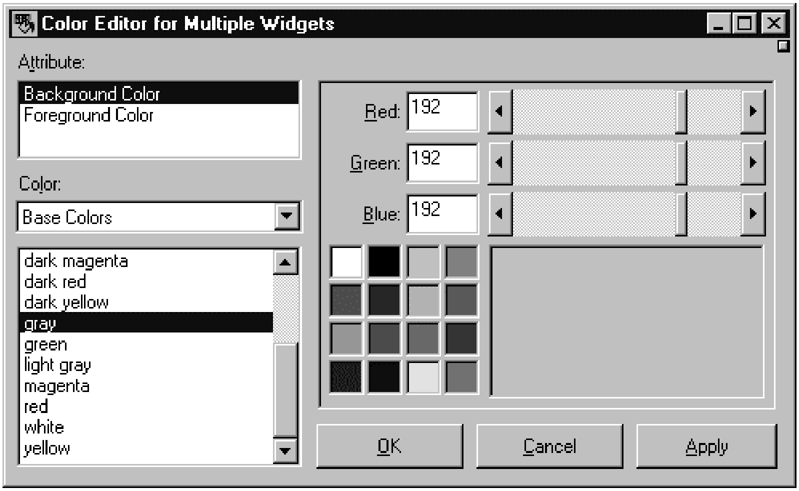
The quick brown fox jumped over the lazy sleeping dog

Bitmap font

Proportional

Colors...

Displays a Colors dialog for the currently selected window or widget(s).



Color Editor for Multiple Widgets

Attribute:

Background Color
Foreground Color

Color: Base Colors

dark magenta
dark red
dark yellow
gray
green
light gray
magenta
red
white
yellow

Red: 192
Green: 192
Blue: 192

The Color Editor provides a list of color attributes (generally just “Background Color” and “Foreground” color). When a color attribute is selected, the name of the color will be selected in the color list and the RGB values and a rendering of the color will appear in the RGB color editor on the right.

To set a color, select it from the color list or use the RGB color editor’s scrollbars or entry fields to set the red, green and blue values of the desired color. As you set one of the RGB values, the system will attempt to find the closest matching color in the color list for you.

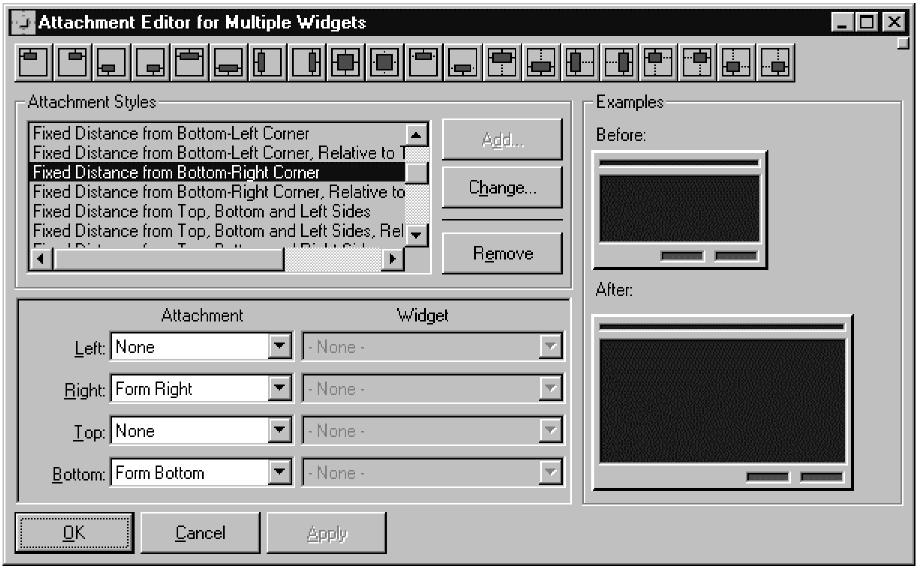
The Color Editor provides three different color lists: “Base Color”, “R3 Color” and “R4 Color”. The first list provides the standard 16 colors supported by the default palettes on all platforms. The other lists are colors supported under various versions of UNIX (for example, R3 & R4). The VisualAge color model, like the windowing system, is based on UNIX Motif. The list of supported colors varies by platform and video driver capabilities. If you select a color that is not in the Base Color list, the operating system tries to match that color to the best of its ability. Windows responds to a request for an unsupported color by displaying the closest color from its default palette. OS/2 responds by attempting to reproduce the desired color through dithering colors in its palette that it is capable of displaying.

The first item in the color list will always be “<Default>”. Selecting this color will set the widget to use its own default colors for that attribute. When a widget has been defined to use its own default color, no color attribute code will be generated.

Attachments...

Displays an attachment editor for the currently selected widget.

This editor is used to specify constraints on a widget. Essentially, by specifying widget attachments you specify what happens when the application window is resized. You can specify all kinds of attachments. Attachments with respect to the application window, attachments with respect to any other widget in the form. Proportional attachments and fixed size attachments can also be specified. A comprehensive list of attachment styles are provided. By default the widget is constrained with respect to the application window (its top left corner). You can also specify any other widget in the window relative to which you want to constrain a particular edge.



Attachment Editor for Multiple Widgets

Attachment Styles

- Fixed Distance from Bottom-Left Corner
- Fixed Distance from Bottom-Left Corner, Relative to T
- Fixed Distance from Bottom-Right Corner
- Fixed Distance from Bottom-Right Corner, Relative to
- Fixed Distance from Top, Bottom and Left Sides
- Fixed Distance from Top, Bottom and Left Sides, Rel

	Attachment	Widget
Left:	None	- None -
Right:	Form Right	- None -
Top:	None	- None -
Bottom:	Form Bottom	- None -

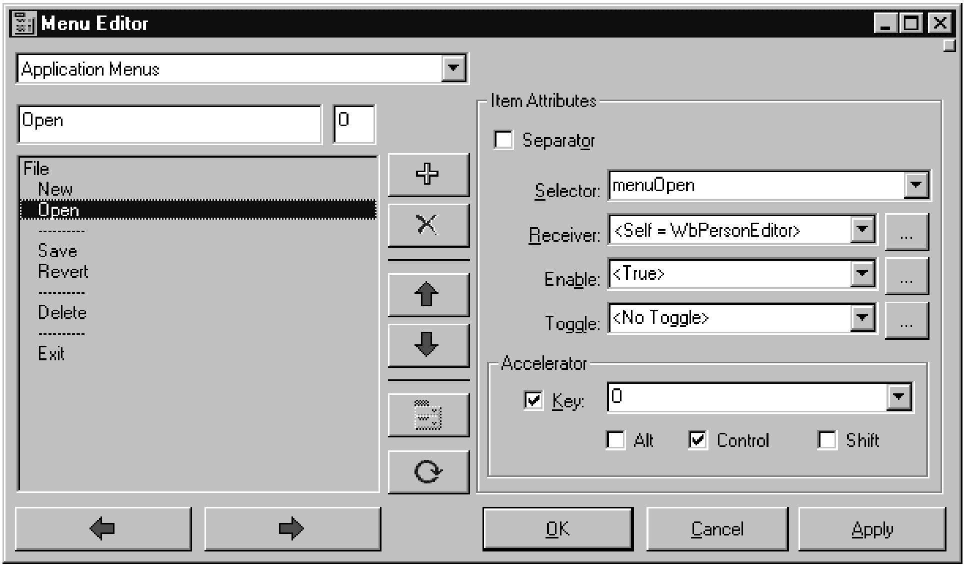
The Attachments command can set the attachments for multiple subpanes at once; add all the desired widgets to the selection before pressing the Attachments button. For more information on multiple selection, see *Selecting Multiple Widgets*. If the **Update Outboards** option is on, the Attachment Editor may be left open. As widgets are selected in the main editor, their current settings will be reflected in the Attachment Editor. They may then be changed and those changes may be locked in via the Apply button.

New with V4.0, the Attachment Editor sports thumbnail before and after views showing the effects of the chosen attachment style on the selected widgets. Widgets can be selected by clicking on their thumbnail representations.

There are hundreds upon hundreds of attachment combinations that can be specified. In reality, only a few of the possible combinations are useful on a frequent basis. Recognizing this, the Attachment Editor provides a fast path to all of the common attachment combinations or styles. The toolbar and listbox at the top of the Attachment Editor provide access to all of the common attachment styles. As you select a style, the individual setting for each side are reflected in the combo boxes below. Locking a button to the lower right corner of the window—an operation that would normally take four steps—can be accomplished in one step by selecting the "Fixed Distance from Bottom-Right Corner" style or selecting the appropriate toolbar button.

Menus...

Displays a Menu Editor for the window.



Menu Editor

Application Menus

Open 0

File
New
Open

Save
Revert

Delete

Exit

Item Attributes

☐ Separator

Selector: menuOpen

Receiver: <Self = WbPersonEditor>

Enable: <True>

Toggle: <No Toggle>

Accelerator

☑ Key: O

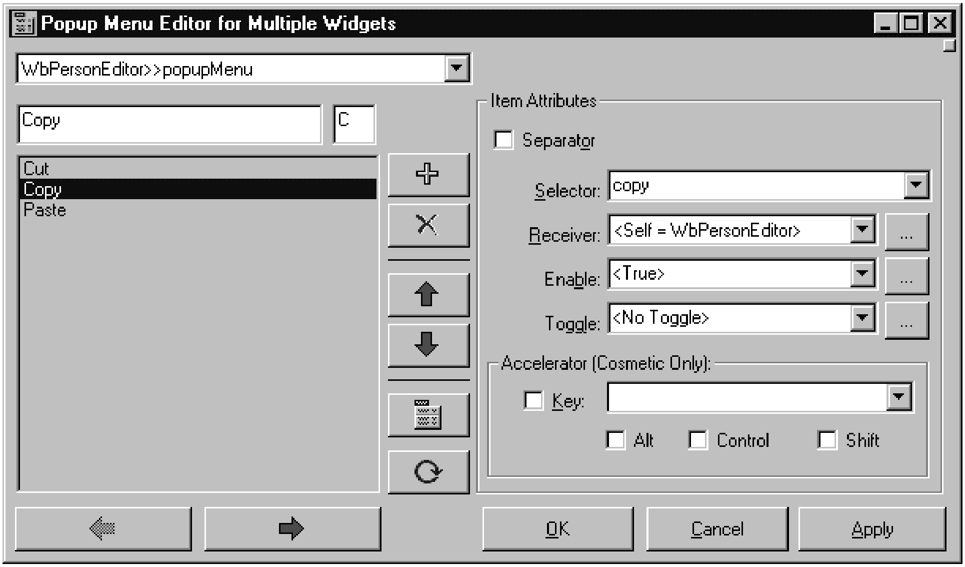
☐ Alt ☑ Control ☐ Shift

The menu editor is used to defined the application's pull down menus. The menu editor allows you to define standard left menus (for example, File, Edit), application menus and standard right menus (for example, Windows, Help). This makes subclassing easy where you keep the left and right menus unchanged and vary the right menus.

The menu editor is a convenient way for specifying menu titles, menu items, menu selectors, menu item separators, menu accelerator key and so on. Cascading of menus is a snap. You can specify a unary menu message selector or a VisualAge callback-style 3 argument selector. Your choice.

In addition to the selector, you can also specify the receiver of the selector (the application by default) as well as whether the menu should be enabled and/or be a toggle menu. For the enable and toggle options, you can optionally enter a method selector (in the same class as the receiver) that should (at runtime) answer true or false to specify the current state of the menu. By setting up these boolean method selectors, you can make menu management a snap at runtime.

Accelerator keys may also be established that use either the **Alt**, **Control** or **Shift** keys (or any combination). The mnemonic key associated with a menu (the underlined letter) can be specified in the field directly to the right of the menu label field.



Popup Menu Editor for Multiple Widgets

WbPersonEditor>>popupMenu

Copy | C

Cut
Copy
Paste

Item Attributes

☐ Separator

Selector: copy

Receiver: <Self = WbPersonEditor>

Enable: <True>

Toggle: <No Toggle>

Accelerator (Cosmetic Only):

☐ Key:

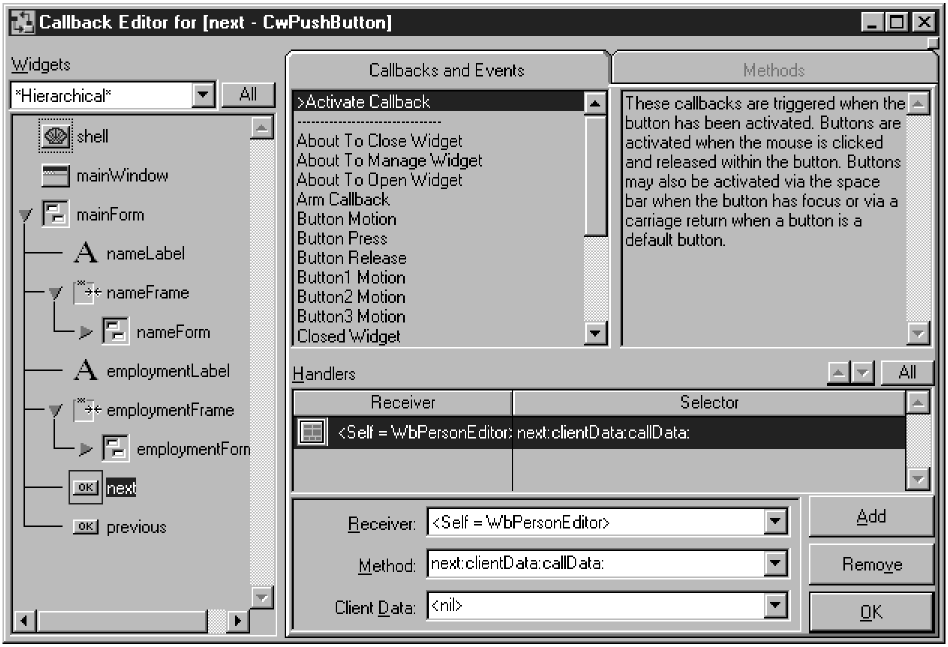
☐ Alt ☐ Control ☐ Shift

Most widgets can have popup menus associated with them. The process of creating a popup menu for a widget is the same as creating a menubar for a window. There are two minor differences:

- When you create a menubar, the top level menu items are menu titles that display horizontally on the menubar. When you create a popup menu, the top level menu items display vertically when the user right-clicks on the widget associated with the menu.
- When you test a menubar, a new window launches with a working example of your menu in it. When you create a popup menu, the menu pops up by itself, without its associated widget.

Callbacks...

Displays a Callback Editor for the currently selected window or widget.



These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Receiver	Selector
<Self = WbPersonEditor:	next:clientData:callData:

Receiver: <Self = WbPersonEditor>

Method: next:clientData:callData:

Client Data: <nil>

This is the editor that you will use to specify callback methods when a specific action is performed on a widget. A callback is a mechanism by which the application is notified when some higher level action is performed on a widget. For example the Activate Callback is used to inform the application that a CwPushButton has been pressed and released. The destroy callback is used to inform the application that a widget has been destroyed. The application can take some appropriate action via the callback method in response to this action. The Callback editor can be invoked from the toolbar, the Attributes menu or by right **double-clicking** on any widget.

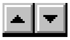
The Callback Editor lists the callbacks that are supported by the widget in question. For example, if you were to bring up a Callback editor on the extended widget CwObjectList, you would see these callback names (along with several others):

Browse Selection Callback
Default Action Callback
Single Selection Callback
Multiple Selection Callback
Extended Selection Callback

Let’s consider Default Action Callback. There is a description provided within the editor that describes when this situation would occur. This default action callback is triggered when you double click on an item on the CwObjectList widget. Triggering a callback? What does that mean? Well, that means the application is given a chance to respond to this user action. For example, if CwObjectList were displaying a list of classes, double clicking on say the class CwBasicWidget, could bring up a class browser on CwBasicWidget. The Smalltalk code that brings up the class browser is a user written piece of code that resides in the callback method for the Default Action Callback.

What is this callback method or where do you specify it? The callback method is a 3 argument method, and you specify it in the combo box entry beside the caption that says **Method**. Actually what you specify is a 3 argument message selector such as `#browseClass:clientData:callData`. New with V4.0, zero (unary) or one argument callback methods are supported for all widgets.

You express the intent that you want to handle the Default Action Callback, by simply selecting the callback name from the list of callbacks and clicking on the **Add** button. Alternatively you can simply double click on the callback name. This will add the method to the list of callback handlers. It is a feature of VisualAge that triggering a single callback can cause multiple methods to be executed. Therefore, you can add as many handlers as you wish. They will fire in the chronological order in which you specified the handlers (this order can be rearranged using the up and down buttons).



By default the receiver of the callback method is **the** application. The application is the class in which the screen is saved into. This class is either directly or indirectly a subclass of WbApplication. However, the receiver of the callback method **need not be** the application. It can be the widget itself which caused the callback to be triggered. Or any other widget that is part of the application. In addition, the receiver can also be any instance variable of the application. This includes local instance variables as well as inherited instance variables.

The 3 argument callback selector, mentioned above, is specified by the user in the Method combo box field. The message selector is typically of the form:

```
#browseClass:clientData:callData:
```

As a convenient user interface feature, WindowBuilder Pro waits for you to start typing a selector name, and as soon as you type the second colon (:) character, it immediately auto-fills the rest of the selector name by automatically appending the text: “`clientData: callData:`”. The last two keyword names `clientData:` and `callData:` are VisualAge naming conventions for callback method selector names.

Zero (unary) and one-argument callbacks handlers are also supported. Leave the colons off, if you do not want the method name expanded to the standard three argument style.

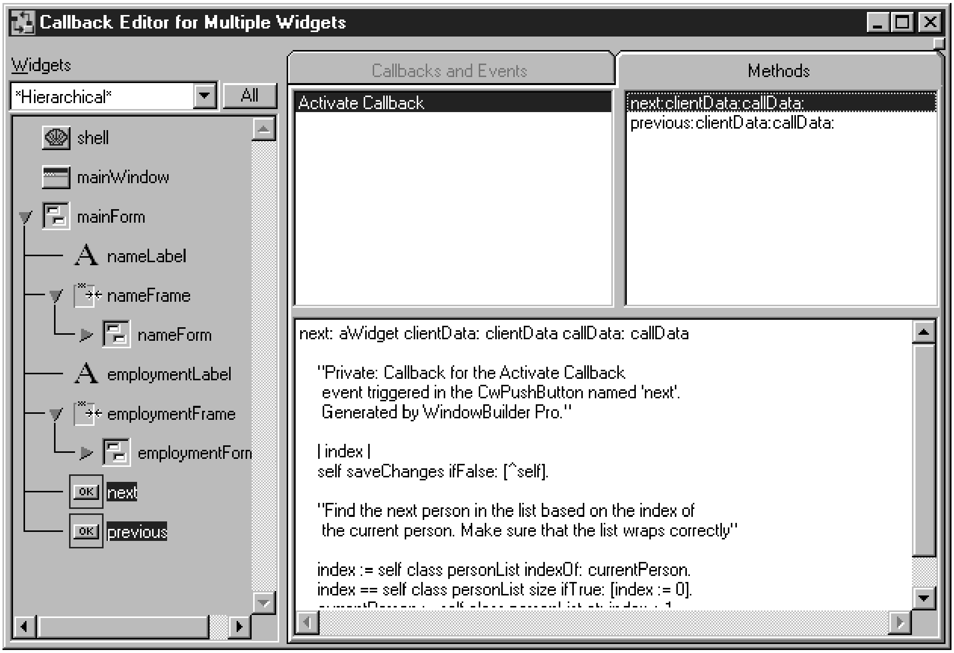
Unary callback handlers do not pass along any arguments and thus do not allow client data to be specified.

For one-argument callbacks, the default argument that is passed is the originating widget. Specifying a unary selector as the client data will cause the attribute of the originating widget specified by that selector to be passed as the argument (e.g., specifying `#selectedItem` as the client data for the Single Selection Callback of a listbox will cause the selected item to be passed as the argument).

When you specify a callback such as the one above for Default Action Callback, WindowBuilder Pro generates a stub method in your application class that looks like this:

```
browseClass: aWidget clientData: clientData callData: callData
   “Private: Callback for the Default Action Callback event
   triggered in the CwObjectList named ‘aCwObjectList’.
   Generated by WindowBuilder Pro.”
```

aWidget is the widget which caused this callback to occur. clientData is any application-specific data you might choose to send. (This is actually specifiable in the callback editor). callData is a widget-specific object whose contents depend upon the widget and the nature of the callback. WindowBuilder Pro also provides a code generation option for annotating the callback method with a more detailed explanation of each of the arguments (especially the callData argument).



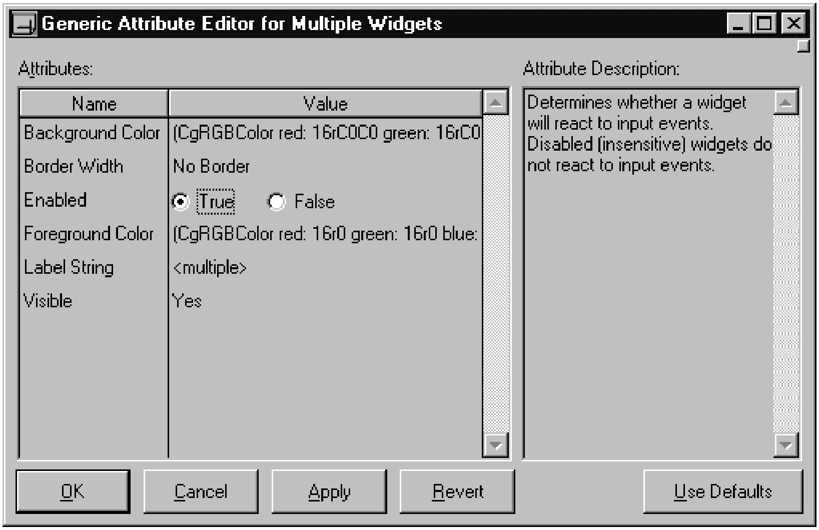
```
next: aWidget clientData: clientData callData: callData

  "Private: Callback for the Activate Callback
   event triggered in the CwPushButton named 'next'.
   Generated by WindowBuilder Pro."

  | index |
  self saveChanges ifFalse: [^self].

  "Find the next person in the list based on the index of
   the current person. Make sure that the list wraps correctly"

  index := self class personList indexOf: currentPerson.
  index == self class personList size ifTrue: [index := 0].
```

The callback handler stub may be viewed and edited using the embedded code browser. Just click on the Methods tab to access the embedded code browser. Click on any widget or group of widgets and the list of callbacks that have handlers will be shown in the upper left list. Select one or more callbacks and view the callbacks handlers that have been defined for them. Clicking on any of the callback handlers (methods) will display its code in the text pane at the bottom of the window. Right-clicking on either the methods list or the text pane will popup the standard menus that you would expect to find in any of the system code browsers. Code may be viewed and edited here. Methods may be deleted, added, new editionss loaded, etc.

Selected Widgets...

Displays the attribute editor for the currently selected window or widget(s). Each attribute editor is described in the Widget Encyclopedia in conjunction with the widget type it edits.

If a heterogeneous collection of widgets is selected, the Generic Attribute Editor will open rather than a widget specific attribute editor. If the CTRL key is held down while invoking this command, the Generic Attribute Editor will also appear. If the Use Generic Editors property is turned on, the Generic Attribute Editor will become the default editor in all cases.



Generic Attribute Editor for Multiple Widgets

Attributes:

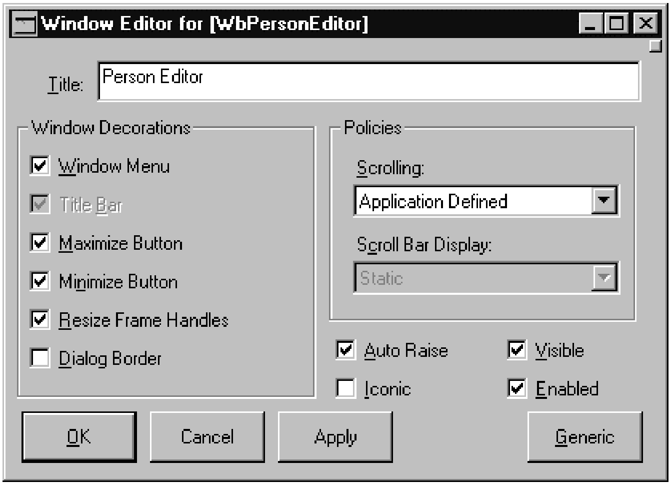
Name	Value
Background Color	(CgRGBColor red: 16rC0C0 green: 16rC0
Border Width	No Border
Enabled	● True ○ False
Foreground Color	(CgRGBColor red: 16r0 green: 16r0 blue:
Label String	<multiple>
Visible	Yes

Attribute Description:

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Window...

Displays a window editor for the currently selected window.



Title: Person Editor

Window Decorations
- ☑ Window Menu
- ☑ Title Bar
- ☑ Maximize Button
- ☑ Minimize Button
- ☑ Resize Frame Handles
- □ Dialog Border

Policies

Scrolling: Application Defined

Scroll Bar Display: Static

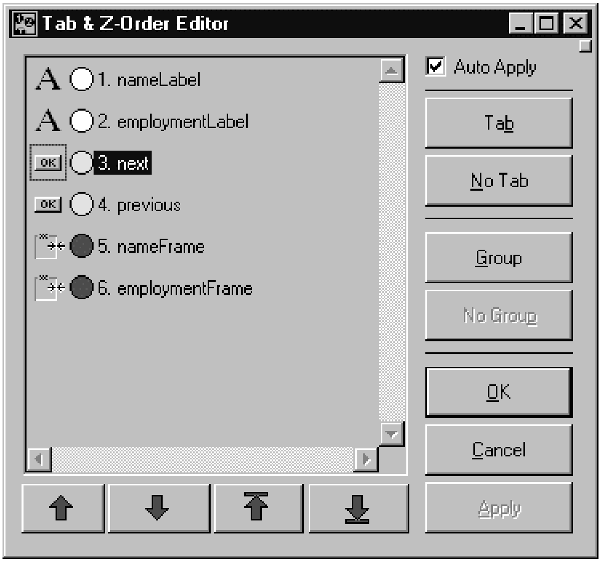
- ☑ Auto Raise
- ☑ Visible
- □ Iconic
- ☑ Enabled

In addition the window's title, its border decorations can be specified by the check boxes on the right (system menu, titlebar, minimize and maximize buttons, resize border and dialog border). Some of the check boxes are mutually exclusive and can only be used in specific combinations. The exact combinations are platform specific and WindowBuilder Pro does its best to support these platform specific combinations (for example, the title bar can be deleted under OS/2, but not under Windows).

You can also specify whether you want the window to automatically include scrollbars or not. This can be set via the comboboxes on the right. Setting the **Scrolling** field to "Automatic" adds scrollbars to the window. The visibility of the scrollbars can then be set via the **Scroll Bar Display** field to either "Static" (always there) or "As Needed" (only visible when the window is too small to display its contents).

Tab & Z-Order...

Displays the Tab & Z-order Editor.



☑ Auto Apply

A ○ 1. nameLabel
A ○ 2. employmentLabel
○ 3. next
○ 4. previous
● 5. nameFrame
● 6. employmentFrame

The Tab & Z-order Editor allows you to change the relative depth of each widget in the Z-order as well as specify whether a widget should be a tab stop and/or a tab group. The listbox in the editor contains a list of all of the widgets visible at the current editing level within the editor (e.g., direct editing a form takes you down a level). The Z-order of each widget is indicated by the number to the left of its name. Its tab status, if any, is indicated to the left of the number. Widgets that are both tab stops and tab groups are indicated by a red circle. Widgets that are tab stops but not tab groups (e.g., buttons in a single tab group) are indicated by a yellow circle. Widgets that are not tab stops are indicated by a white circle.

The ⇧, ⇩, ⤒ or ⤓ buttons may be used to reorder the selected widget or widgets. If a discontinuous collection of widgets is selected, they will all be collected together and follow the movement of the first widget in the selection.

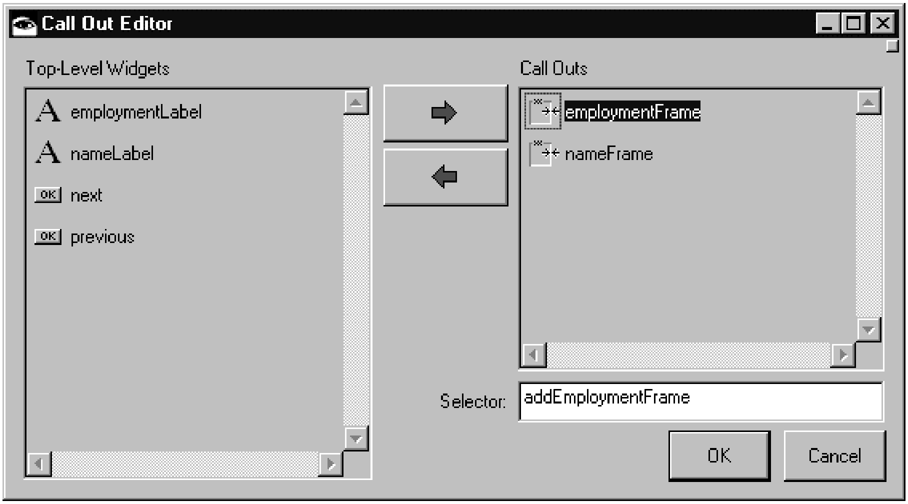
The **Tab** and **No Tab** buttons are used to specify whether the selected widgets are to be tab stops or not. The **Group** and **No Group** buttons determine whether the widgets are tab groups as well. At any level, there may be one and only one tab group. If you need to establish more than one, you should put each widget group into its own CwForm or CwRowColumn.

The interaction been tab stops, tab groups and the initial tab stop is a bit complicated. The initial focus goes to the first widget that is a tab stop and is not a tab group (for example, a button that is part of the top level tab group). If all widgets are tab groups, focus goes to the first widget that is a full tab stop (e.g., both a tab stop and a tab group like a CwText). Widgets which do not receive focus are not part of the tab order, whereas they are part of the z-order (e.g., CwLabels and CwSeparators).

If the **Auto Apply** check box is set, all changes made in the editor will be immediately reflected in the editing window. If it is not set, changes are batched up and applied when either the **Apply** or **OK** buttons are pressed.

Call Outs...

Displays the Call Out Editor.

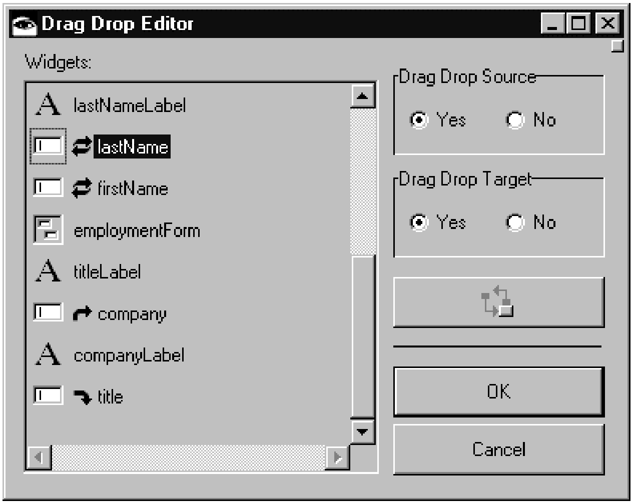


Selector: addEmploymentFrame

The Call Out Editor gives you the opportunity to exercise more control over how your window definition code is factored. For a large window with lots of widget definitions, WindowBuilder Pro will generate a very large `#addWidgets` method. The Call Out Editor allows you to have any top level widget (e.g., child of the main form) generated into its own method. If a CwForm, CwRowColumn, CwScrolledWindow, Frame or Notebook widget is so designated, it and all of its nested children will be generated in the specified method. The editor interface is simple. On the left side of the window is a list of all top level widgets. The ⇨ button moves one or more widgets to the right hand list where a call out method selector may be specified. Note that this should be unary selector (no arguments) and that a default method name is built for you based on the widget’s name. The ⇦ button may be used to transfer a widget back to the left hand list where it will be generated in the main `#addWidgets` method.

Drag Drop...

Displays the Drag Drop Editor.



Drag Drop Editor

Widgets:

- lastNameLabel
- ⇄ lastName
- ⇄ firstName
- employmentForm
- titleLabel
- ↱ company
- companyLabel
- ↴ title

Drag Drop Source: ● Yes ○ No

Drag Drop Target: ● Yes ○ No

The Drag Drop Editor allows you to specify whether a widget should be a drag drop source or target. If a widget is defined to be a drag drop source, an EwSourceAdapter will be created for it. If a widget is defined to be a drag drop target, an EwTargetAdapter will be created for it. Widgets that are either drag drop sources or targets are identified via the following symbols:

↱ Drag Drop Source

↴ Drag Drop Target

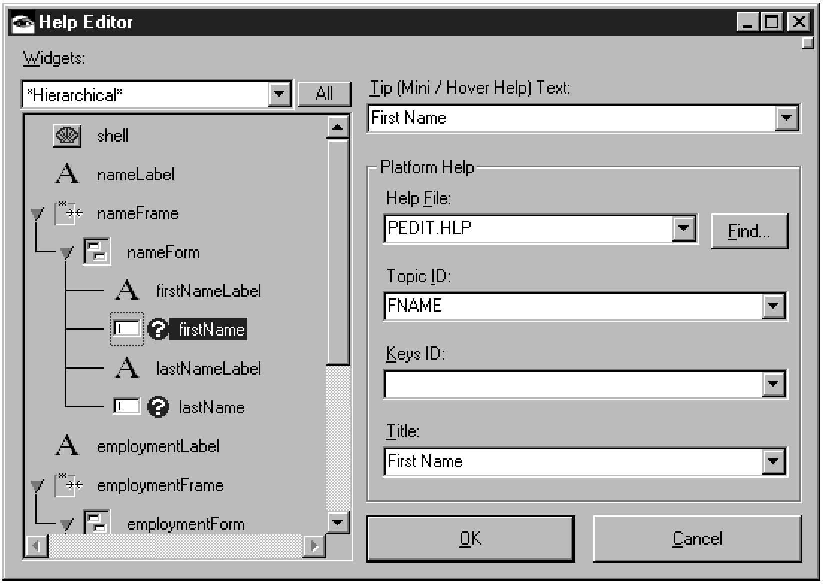
⇄ Drag Drop Source and Target

The Drag Drop Editor also allows you to specify callback handlers for any drag drop adapter by clicking on the Callback button.

A number of different drag drop events are supported. The class WbDragDropExample provides an example of how to use the various drag drop events.

Help...

Displays the Help Editor.



Widgets: *Hierarchical*

Tip (Mini / Hover Help) Text: First Name

Help File: PEDIT.HLP

Topic ID: FNAME

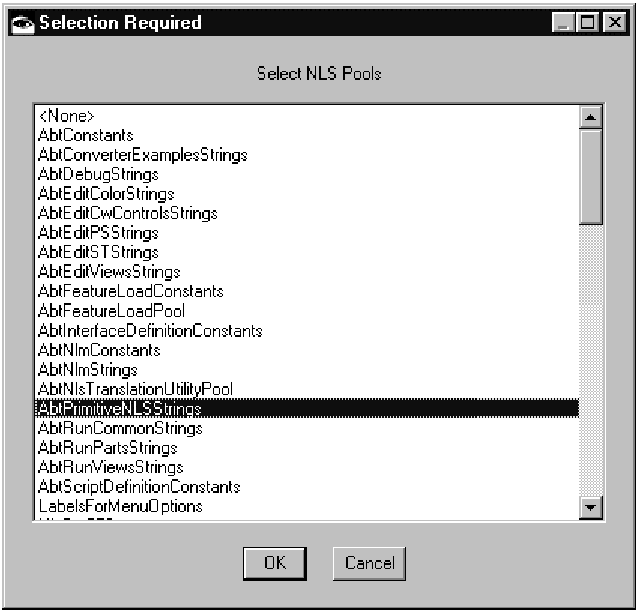
Keys ID:

Title: First Name

The Help Editor allows you to specify tool tips (e.g., hover/mini help) and platform help (help file, topic ID, etc.) for any widget. Widgets that have help (hover help or platform help) are identified by the ❓ symbol.

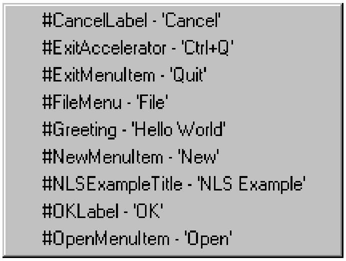
NLS...

Displays the NLS Editor.



<None>
AbtConstants
AbtConverterExamplesStrings
AbtDebugStrings
AbtEditColorStrings
AbtEditCwControlsStrings
AbtEditPSStrings
AbtEditSTStrings
AbtEditViewsStrings
AbtFeatureLoadConstants
AbtFeatureLoadPool
AbtInterfaceDefinitionConstants
AbtNlmConstants
AbtNlmStrings
AbtNlsTranslationUtilityPool
AbtPrimitiveNLSStrings
AbtRunCommonStrings
AbtRunPartsStrings
AbtRunViewsStrings
AbtScriptDefinitionConstants
LabelsForMenuOptions

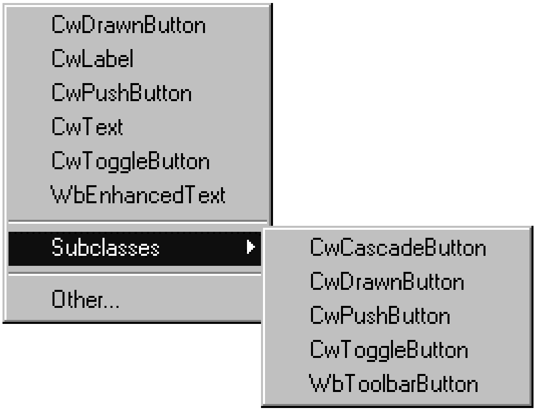
Text labels for buttons, labels, window titles and menus can be specified as NLS pool dictionary keys rather than strings. Use the NLS Editor to assign one or more NLS pool dictionaries to the currently edited class (see the *IBM Smalltalk Programmer's Reference* for instructions for creating NLS pool dictionaries). Typing in a "#" followed by the pool key name sets the text of the widget to that pool constant. The actual string that is held by the pool dictionary will be displayed in the editing window. **Right-clicking** on any text field (e.g., the Label String field in the CwPushButton editor) will pop up a menu from which an NLS pool key may be selected. If multiple NLS pools are assigned to the class, this popup menu will have multiple cascading entries. When WindowBuilder Pro generates the code for the window, the appropriate NLS pool key is generated rather than the actual text as seen in the widget.



Morph...

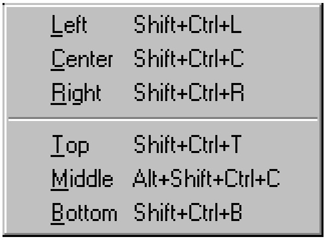
Quickly changes - “morphs” - any widget from one type to another. For example, a WbScrolledList instance could be converted into a CwComboBox or WbRadioBox. Common attributes are automatically translated. Attributes not needed by the target class are lost. Attributes not provided by the source class are defaulted.

Right-clicking on the **Morph** button in the toolbar will popup a menu containing all of the lists morphing types for the selected widget. If a widget class has subclasses, all of those are listed as well.



Warning: Some care is needed on your part when morphing a widget into a radically different type. WindowBuilder Pro will map over any attributes the two have in common as well as event callbacks for any shared events. You must be careful that the event callbacks do, in fact, make sense for the new type. For example, a Default Action Callback for a CwText would not be appropriate for an CwPushButton. It is recommended that morphing be limited to similar classes of objects.

Align



Left

Aligns the left side of the selected widgets to the first widget in the selection.

Center

Aligns the selected widgets so that one vertical axis goes through all their centers.

Right

Aligns the right side of the selected widgets to the first widget in the selection.

Top

Aligns the top of the selected widgets to the first widget in the selection.

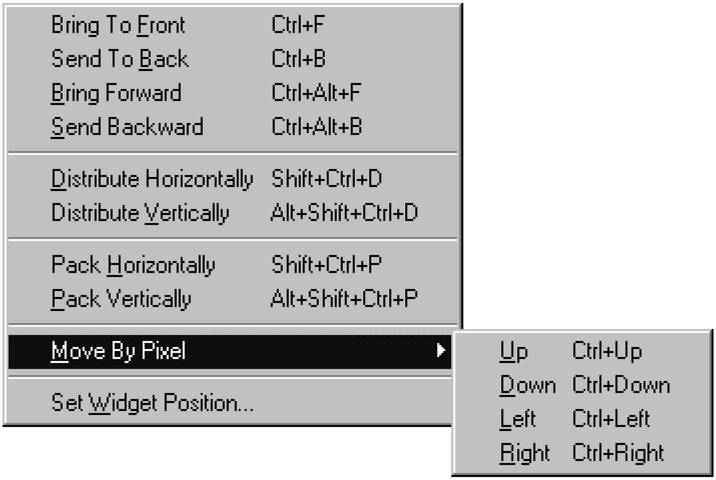
Middle

Aligns the selected widgets so that one horizontal axis goes through all their centers.

Bottom

Aligns the bottom of the selected widgets to the first widget in the selection.

Position



Bring To Front

Moves the selected widget in front of any widgets that conceal it. This places the widget at the top of the z-order. If the widget is also a tab stop, it will become the first tab stop on the window.

Send To Back

Moves the selected widget behind any widgets it overlaps. This places the widget at the bottom of the z-order. If the widget is also a tab stop, it will become the last tab stop on the window.

Bring Forward

Moves the selected widget forward one position in the z-order. This command also moves it forward in the tab order if the widget is a tab stop.

Send Backward

Moves the selected widget backward one position in the z-order. This command also moves it backward in the tab order if the widget is a tab stop.

Distribute Horizontally

Evenly distributes the selected widgets horizontally, i.e. leaving the first and last selected panes in the same location, forces the horizontal space between each pane to be the same.

Distribute Vertically

Evenly distributes the selected widgets vertically, i.e. leaving the first and last selected panes in the same location, forces the vertical space between each pane to be the same.

Pack Horizontally

Packs the selected widgets so that they are one grid spacing apart horizontally.

Pack Vertically

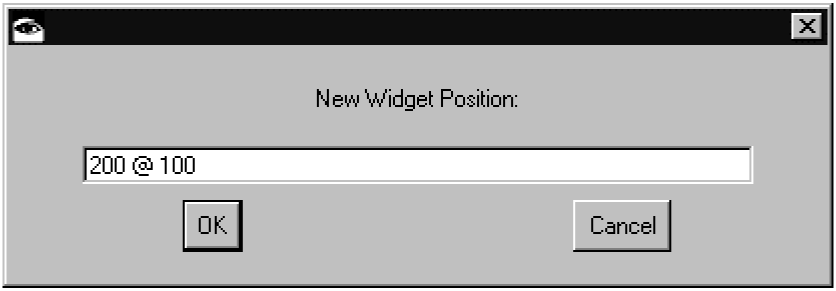
Packs the selected widgets so that they are one grid spacing apart vertically.

Move By Pixel

Moves the selection one pixel in the direction specified. These commands are replicated in the floating Nudge tool available from the Options | Tools menu.

Set Widget Position...

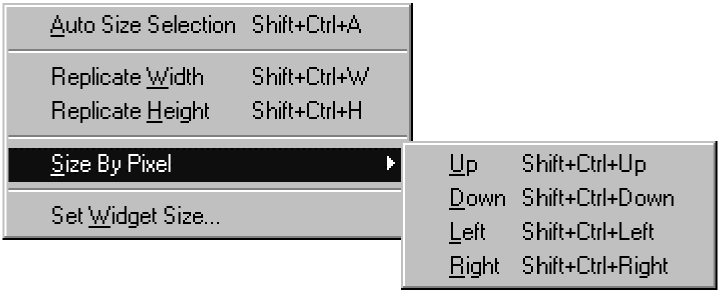
Sets the location at which the currently selected widget will initially be placed.



New Widget Position:

200 @ 100

Size



Auto Size Selection

Sets the size of the selected widget to the size specified by the widget’s answer to the `#preferredExtent` message. The default suggested size is the current size of the widget, i.e. a no-op. This command is useful for sizing simple widgets like labels without much effort.

Replicate Width

Sets the width of all widgets in the selection to the width of the first widget in the selection.

Replicate Height

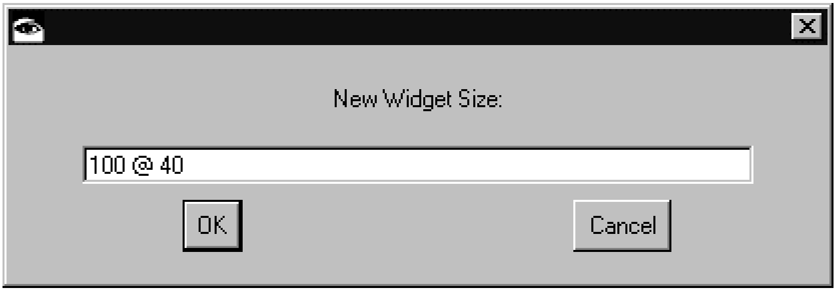
Sets the height of all widgets in the selection to the height of the first widget in the selection.

Size By Pixel

Sizes the selection one pixel in the direction specified. When sizing, the top left corner of the selection remains in place, and the lower right shifts. These commands are replicated in the floating Nudge tool available from the Options | Tools menu.

Set Widget Size...

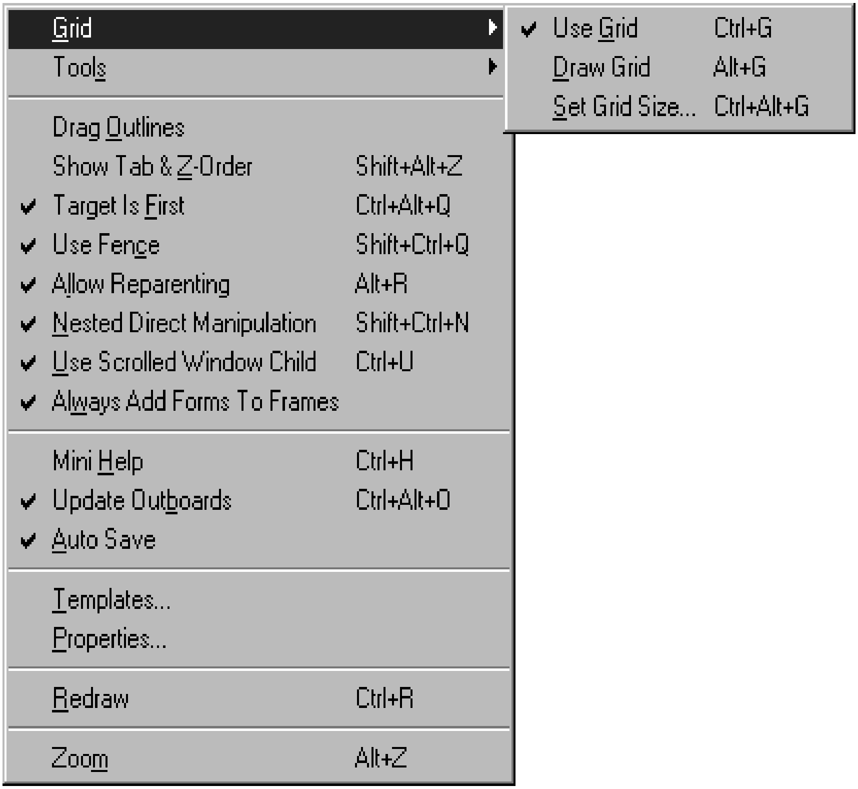
Displays a dialog in which you can set the size of the selected widget to a specific pixel size.



New Widget Size:

100 @ 40

Options


Grid
Tools
Drag Outlines
Show Tab & Z-Order Shift+Alt+Z
✓ Target Is First Ctrl+Alt+Q
✓ Use Fence Shift+Ctrl+Q
✓ Allow Reparenting Alt+R
✓ Nested Direct Manipulation Shift+Ctrl+N
✓ Use Scrolled Window Child Ctrl+U
✓ Always Add Forms To Frames
Mini Help Ctrl+H
✓ Update Outboards Ctrl+Alt+O
✓ Auto Save
Templates...
Properties...
Redraw Ctrl+R
Zoom Alt+Z
✓ Use Grid Ctrl+G
Draw Grid Alt+G
Set Grid Size... Ctrl+Alt+G

Use Grid

Toggles the grid function on or off. When set on, an invisible grid is overlaid on the design surface that widgets snap to when being placed, dragged, or sized.

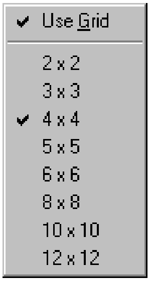
Draw Grid

Toggles the grid display function on or off. When set on, WindowBuilder Pro displays the grid.

Set Grid Size...

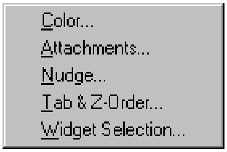
Displays a dialog that allows you to set the grid size of the design surface in the x and y directions.

Right-clicking on the **Grid** button in the lower right corner of the WindowBuilder Pro window pops up a menu from which you can select an appropriate grid size. You can also specify whether the grid should be used or not.



Tools

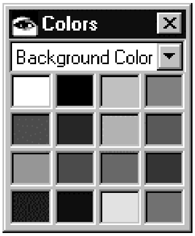
Displays a submenu with the following choices:



WindowBuilder Pro remembers the status (e.g., size, position and open/close) of each floating tool when it is closed. Any tools left open when WindowBuilder Pro closes will be automatically opened when WindowBuilder Pro is restarted.

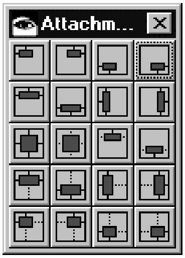
Color

Open the floating color tool. This is tool can be used to set the colors for the selected widgets without the need to open the Color Editor.



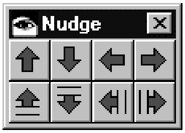
Attachments

Open the floating attachment tool. This is tool can be used to set the attachments for the selected widgets without the need to open the Attachment Editor. Each button in the tool represents a different attachment style. See the discussion of the Attachment Editor for a discussion of these styles.



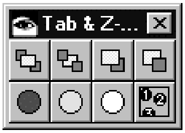
Nudge

Open the floating nudge tool. This tool is used for fine-tuning the position and size of the selected widgets. The top row of buttons are used to position a widget in one pixel increments. The bottom row of buttons are used to size a widget in one pixel increments. These functions replicate the **Move By Pixel** and **Size By Pixel** functions found on the Position and Size menus respectively.



Tab & Z-Order

Open the floating tab & z-order tool. This tool is used for fine-tuning the tab and z-order of the selected widgets. The top row of buttons are used to change the tab and z-order of the selected widgets. These functions replicate the **Bring To Front**, **Send To Back**, **Bring Forward** and **Send Backward** functions found on the Position menu.

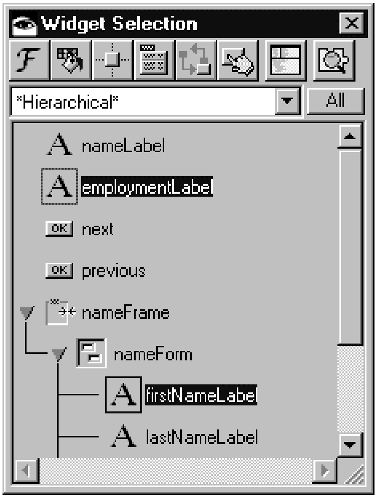


The first three buttons on the bottom row are used to change the tab stop status of the selected widgets. The first button (red) sets the selected widgets to be full tab stops.

The second button (yellow) sets the selected widgets to be tab group members. The third button (white) sets the selected widgets not to be tab stops. The last button on the bottom row are used to toggle on and off the display of the tab and z-order tags. This is the same as the **Show Tab & Z-Order** option on the Options menu.

Widget Selection

Open the floating widget selection tool. This tool can be used to select any combination of widgets on the design surface. Unlike the main design surface, widget selection is not limited to widgets within the same parent group. This allow you to change the attributes of widgets at different levels in the widget hierarchy simultaneously. **Double-clicking** a widget will allow you to edit the widget's name. The toolbar includes all of the standard attribute setting commands like **Font**, **Color**, **Attachments**, **Menus**, **Callbacks** and **Attributes**. A **Browse Widget Class** button and optional (if the WindowBuilder Pro - Tools configuration is loaded) **Inspect** button are also included.



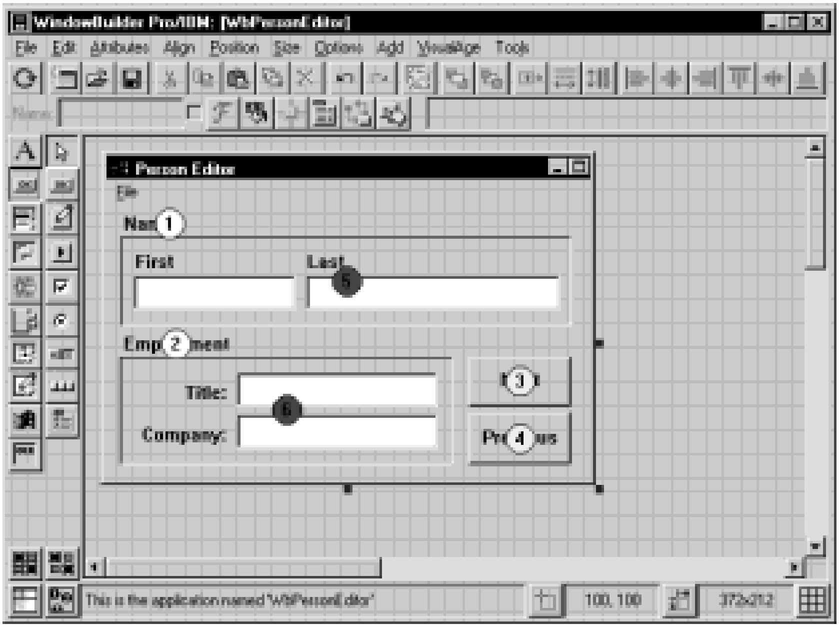
The widget list can display the window's widgets either hierarchically or alphabetically by name or by type. It can also filter the list to show only one type of widget at a time. The **All** button in the upper right corner makes it easy to select all of the widgets in the window or all the widgets of a particular type.

Drag Outlines

Toggles the outline drag function on and off. When set on, widgets will be represented as dotted outlines when they are dragged around the screen. The default is off in which case the widgets themselves are dragged around the screen.

Show Tab & Z-Order

Toggles the Tab & Z-order display function on and off. When set on, a filled circle containing a widget's Tab & Z-order will be displayed on top of the widget. The color of the circle indicates the widget's status as a tab stop. Widgets that are both tab stops and tab groups (e.g., full tab stops) are indicated by red. Widgets that are tab stops but not groups (e.g., buttons in a tab group) are indicated by yellow. Widgets that are not tab stops at all are white (e.g., labels and separators).


WindowBuilder Pro/IBM: [WbPersonEditor]
File Edit Attributes Align Position Size Options Add VisualAge Tools
Name:
Person Editor
File
Nam 1
First
Last 5
Emp 2 ment
Title:
Company:
6
3
Pr 4 us
100, 100
372x212

This is the application named 'WbPersonEditor'

When the tab order tags are visible, they may be moved via drag and drop. This provides a convenient mechanism for rearranging the tab order without the need to open the Tab & Z-Order Editor.

Target Is First

Toggles the selection mode between target-is-first mode and target-is-last mode. The latter is the default mode of operation for VisualAge. The target widget is the widget that acts as the model for all multiple widget editing commands.

Use Fence

This specifies whether widgets should be constrained to the bounding box of their parent widget. Turning this off will allow a widget to be positioned anywhere within its parent (even off screen out of view).

Allow Reparenting

Specifies whether widgets may be reparented by dragging them from one parent to another. For example, a EwTableList could be dragged into a CwScrolledWindow to give it scrolling abilities.

Nested Direct Manipulation

Specifies whether VisualAge-style direct manipulation of nested widgets is allowed. Turning on this option will enable direct manipulation of nested widgets and will automatically put the selected widget's parent into direct select mode. Children may be added to a CwForm, CwRowColumn, CwScrolledWindow, Frame or Notebook by dropping the widget within the boundaries of the desired parent. Selecting multiple widgets follows the rule that only widgets at the same level (with the same parent) may be selected simultaneously.

Use Scrolled Window Child

Specifies whether operations on CwScrolledWindows should pass through to the child. For example, should double clicking on a CwScrolledWindow invoke the attribute editor of the CwScrolledWindow or the child? With this enabled, holding the ALT key down when double clicking will open the attribute editor on the CwScrolledWindow itself.

Always Add Forms To Frames

Specifies whether frames should always be created with an embedded form as their primary child. This is more intuitive for most developers who expect to be able to place multiple widgets within a frame (frames are only allowed one direct child).

Mini Help

Toggles the display of popup mini help (balloon/hover help) in all attribute editors. Mini help may be turned on and off in any attribute editor via hitting CTRL+H. If mini help is turned off, hitting F1 in any attribute editor will pop up a mini help window for the widget under the cursor (and only that widget).

Update Outboards

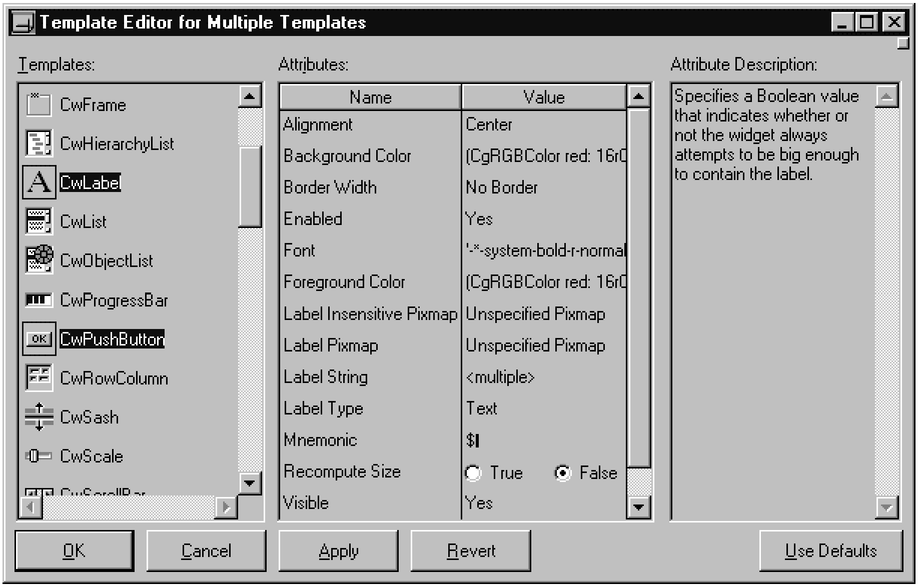
Toggles the outboard update function on and off. When set on, WindowBuilder Pro will automatically update any outboard windows (e.g., attribute editors, attachment editor, etc.) with the currently selected widget or widgets.

Auto Save

Toggles the Autosave function on or off. Autosave prevents WindowBuilder Pro from querying you to save the window's definition every time you wish to test it. This facilitates rapid testing.

Templates...

Displays the template editor.



Template Editor for Multiple Templates

Templates:

- CwFrame
- CwHierarchyList
- CwLabel
- CwList
- CwObjectList
- CwProgressBar
- CwPushButton
- CwRowColumn
- CwSash
- CwScale
- CwScrollBar

Attributes:

Name	Value
Alignment	Center
Background Color	(CgRGBColor red: 16r0
Border Width	No Border
Enabled	Yes
Font	'-*-system-bold-r-normal
Foreground Color	(CgRGBColor red: 16r0
Label Insensitive Pixmap	Unspecified Pixmap
Label Pixmap	Unspecified Pixmap
Label String	<multiple>
Label Type	Text
Mnemonic	$I
Recompute Size	○ True ● False
Visible	Yes

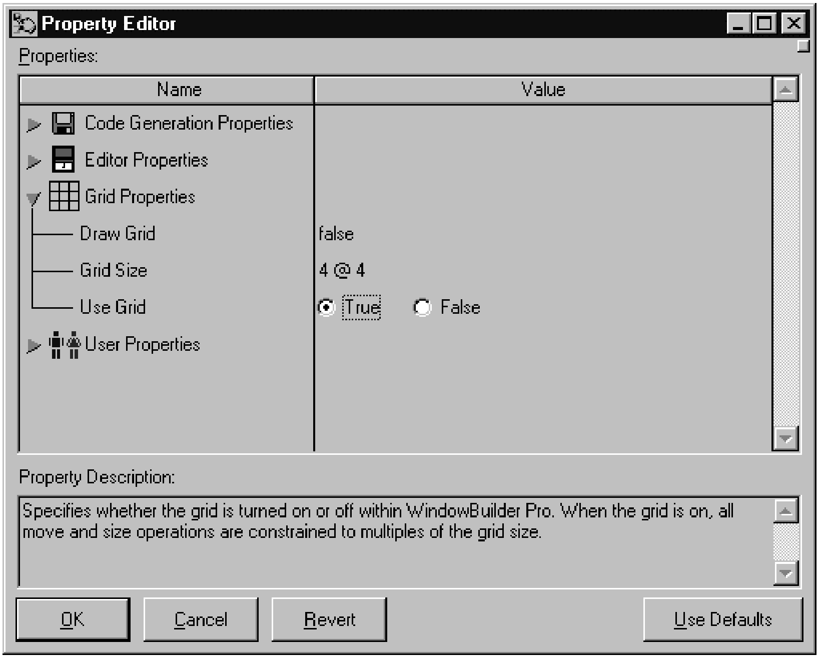
Attribute Description:

Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

The template editor allows you to specify default attribute values for attributes of all the widgets supported by WindowBuilder Pro. These widgets are the concrete classes in the Common Widgets and the Extended Widgets subsystem of VisualAge. Once you change a default value, that new value will be used for all new instances of that widget that you place in the editor. For example, all CwArrowButtons are upward pointing by default. This can easily be changed so that all new buttons are right pointing when first dropped into the editor.

Properties...

Displays the property editor.



Code Generation Properties

Editor Properties

Grid Properties

Name	Value
Draw Grid	false
Grid Size	4 @ 4
Use Grid	● True ○ False

User Properties

Property Description:

Specifies whether the grid is turned on or off within WindowBuilder Pro. When the grid is on, all move and size operations are constrained to multiples of the grid size.

The properties editor is used to customize your WindowBuilder Pro environment. You can customize the code generation properties, the editor properties, grid properties and the user properties.

Allow Reparenting Specifies whether widgets may be reparented by dragging them from one parent to another. For example, a EwTableList could be dragged into a CwScrolledWindow to give it scrolling abilities.

Always Add Forms To Frames Specifies whether frames should always be created with an embedded form as their primary child. This is more intuitive for most developers who expect to be able to place multiple widgets within a frame (frames are only allowed one direct child)..

Auto Save Specifies whether the window definition should automatically be saved before testing a window.

Auto Update Outboards Specifies whether the outboard windows should be updated when new widgets are selected in the main editing window.

Comment Methods Specified whether the comment field of each method should be set to the current copyright string.

Company Name Specifies the user’s company name.

Confirm Non-Undoable Specifies whether the user is asked to confirm non-undoable operations.

Copyright Specifies the copyright string to use in generated code. Automatic word substitution is supported: “%Y” is replaced by the current calendar year. “%C” is replaced by the company name. “%U” is replaced by user name. “%D” is replaced by the current date. “T” is replaced by the current time.

Copyright After Body Specifies whether the copyright string should appear after the method body. When this property is false, the copyright text is generated before the method body right after the method comment. This is only applicable when “Generate Copyright” is set to true.

Default Widget Font Specifies the default font used for all widgets..

Drag Outlines Specifies whether widgets should be dragged as outlines or not.

Draw Grid Specifies whether the grid is visible within WindowBuilder Pro.

Generate All Stubs Specifies whether callback stubs should be generated for all receiver objects. If false, callback stubs are generated only for the current application.

Generate Default Handler Names Specifies whether the Callback Editor should generate default handler names when new callback/event handlers are added.

Generate OLE Properties Specifies whether OLE/ActiveX properties should be generated for OleControl instances. If turned on, each property setting will be generated as “propertyAt: <*Name*> put: <*value*>”. Settable properties are limited to String, Integer, Float and Boolean. If turned off, the property settings will be generated to an OLE file named <*widgetName*>.CON. All properties are settable, but the .CON file must be present in the working directory.

Generate Copyright Specifies whether each generated method should include a copyright string.

Grid Size Specifies the grid size used in the WindowBuilder Pro editor. When the grid is on, all move and size operations are constrained to multiples of the grid size.

Handle Size Specifies the handle size used in the WindowBuilder Pro editor.

Inherit Pool Dictionaries Specifies whether pool dictionary references should be inherited from the superclass. The default is true. For better portability to other Smalltalk dialects, set this to false.

Line Before Comment Specifies whether there should be a blank line between the message pattern and the method comment.

Make Callbacks Private Specifies whether generated callback stubs should be private methods. True indicates that they should be private. False indicates that they should be public.

Max Undo Levels Specifies the maximum number of undo levels maintained by the WindowBuilder Pro editor.

Max Window Size Specifies the maximum window size that can be built using WindowBuilder Pro.

Mini Help Delay Specifies the Mini Help delay in milliseconds. This delay is the time between the pointer passing over a widget and its Mini Help description appearing.

Mini Help Enabled Specifies whether Mini Help should be enabled or not. If true, WindowBuilder will pop up help descriptions of attributed editor items when the cursor is passed over them.

Mini Help On Toolbars Only Specifies whether Mini Help should be available only for toolbars or not. If true, help will only appear for toolbar items. If false, help will appear on all items.

Nested Direct Manipulation Specifies whether VisualAge-style direct manipulation of nested widgets is allowed. Setting this to true will enable direct manipulation of nested widgets and will automatically put the selected widget’s parent into direct select mode.

Show All Errors Specifies whether all internal editor errors should be displayed (for debugging purposes).

Show Attachment Palette Specifies whether the floating attachment palette should be displayed.

Show Color Palette Specifies whether the floating color palette should be displayed.

Show Nudge Palette Specifies whether the floating nudge palette should be displayed.

Show Tab & Z-Order Palette Specifies whether the floating tab & z-order palette should be displayed.

Show Widget Selection Palette Specifies whether the floating widget selection palette should be displayed.

Show Z Order Specifies whether the z-order of the widgets should be shown in the editor.

Target Is First Specifies whether the target widget is the first widget selected in a sequence. Setting this to false, will emulate the VisualAge target-is-last-selected model.

Use Add To Format Specifies whether methods should be generated using AddTo format rather than Add format. In AddTo format, parents are passed in as arguments (e.g., aForm, aMenuBar, etc.). In Add format, parents are in-lined (e.g., self form) and no arguments are passed.

Use Default Code Generation Style Specifies whether the WindowBuilder Pro default code generation style will be used. If this is turned off, the code generator will only use pure CwWidget constructs (no WindowBuilder Pro extensions). The default method created will be called #createWorkRegion and auto-scaling will be disabled.

Use Fence Specifies whether widgets should be constrained to the bounding box of their parent widget. Setting this to false will allow a widget to be positioned anywhere within its parent (even off screen out of view).

Use Generic Editors Specifies whether custom or generic attribute editors should be used by default.

Use Grid Specifies whether the grid is turned on or off within WindowBuilder Pro. When the grid is on, all move and size operations are constrained to multiples of the grid size.

Use Long Callback Comments Specifies whether long callback comments should be generated. Long comments include complete descriptions of each of the callback arguments (e.g., the callData).

User Name Specifies the user’s name.

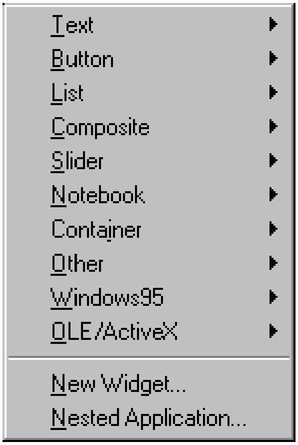
Use Scrolled Window Child Specifies whether operations on CwScrolledWindows should pass through to the child. For example, should double clicking on a CwScrolledWindow invoke the attribute editor of the CwScrolledWindow or the child? With this enabled, holding the ALT key down when double clicking will open the attribute editor on the CwScrolledWindow itself.

Use Side Handles Specifies whether the WindowBuilder Pro editor will display side handles in addition to the corner handles.

Redraw

Forces the window to repaint itself.

Add



Text

Displays a submenu containing all the text display and editing widgets (CwLabel, CwText and WbScrolledText). When a widget type is selected from the submenu, the cursor will be loaded with an example of this widget, which you can then place in the window (i.e. this performs the same function as the Widgets Palette).

Button

Displays a submenu containing all the button-type widgets (CwPushButton, CwDrawnButton, CwArrowButton, CwToggleButton and WbRadioBox).

List

Displays a submenu containing all the list-oriented widgets, or widgets which allow the user to select one from a group of items (WbScrolledList, CwComboBox, EwSpinButton, CwObjectList , CwHierarchyList, EwDrawnList, EwTableList and EwTableTree).

Composite

Displays a submenu containing all composite widgets (CwForm, CwRowColumn, CwFrame, WbFrame and CwScrolledWindow).

Slider

Displays a submenu containing widgets used to set numerical values (EwSlider, CwScrollBar and CwScale).

Notebook

Displays a submenu containing all the notebook-oriented widgets (EwPMNotebook, EwWINNotebook and EwPage).

Container

Displays a submenu containing all the iconic-container widgets (EwIconArea, EwIconList, EwFlowedIconList and EwIconTree).

Other

Displays a submenu containing miscellaneous widgets (CwDrawingArea and CwSeparator).

Windows 95

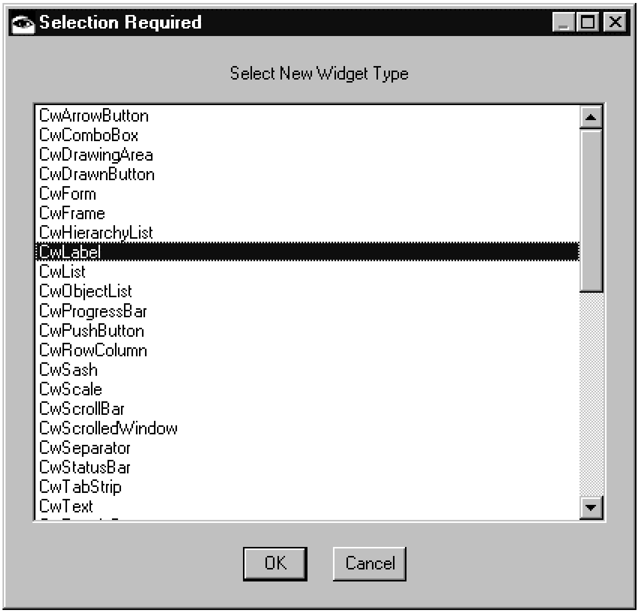
Displays a submenu containing all of the Windows 95 widgets (under Windows 95 or NT 4.0 only).

OLE/ActiveX

Displays a submenu containing OleClient and OleControl.(under Windows 95 or NT 4.0 only). Wrappered OLE/ActiveX widgets (AbtOleExtendedWidget subclasses) will also appear in this list.

New Widget...

Displays a dialog that allows you to specify a widget type by name to place on the editing surface. This command would be useful for adding widgets that are not on the palette.



Selection Required

Select New Widget Type

CwArrowButton
CwComboBox
CwDrawingArea
CwDrawnButton
CwForm
CwFrame
CwHierarchyList
CwLabel
CwList
CwObjectList
CwProgressBar
CwPushButton
CwRowColumn
CwSash
CwScale
CwScrollBar
CwScrolledWindow
CwSeparator
CwStatusBar
CwTabStrip
CwText

Nested Application...

Displays a dialog that allows you to select a WbApplication subclass to embed in the current window. Windows generated using the WindowBuilder Pro application framework can be used both as standalone windows or treated as widgets and embedded within other applications.

Chapter 9 Common Widgets Overview

This chapter introduces the Common Widgets (CW) subsystem of VisualAge. The Common Widgets classes and methods enable programmers to design and build graphical user interfaces using an application programming interface (API) based on OSF/Motif. Using the Common Widgets subsystem, programmers can do the following:

- Create individual widgets, including buttons, lists, text, menus, and dialog boxes
- Create compound widget structures by combining individual widgets
- Specify the positioning of widgets relative to each other
- Program operations to occur in response to user actions

These capabilities are described later in this chapter. In addition, this chapter explains how the system is based on OSF/Motif, gives an overview of the Common Widgets class hierarchy, and describes the basic approach for building an application.

OSF/Motif Compatibility

The Common Widgets subsystem is based on the OSF/Motif C programming interface standard. This section is of interest to developers familiar with Motif. It describes the strategy used to translate Motif C types and functions to Smalltalk classes and methods. Experienced Motif programmers will be able to apply their knowledge of the C programming interface directly when programming Common Widgets.

Smalltalk classes have been created for most C data types. These classes are named by prefixing the Motif data structure name with Cw (after first removing any X, Xt, or Xm prefix). For example, the Motif data structure Widget is represented by the Smalltalk class CwWidget.

Motif functions have corresponding Smalltalk methods. To understand this mapping, consider the function XmListSelectItem below:

```
void XmListSelectItem (widget, item, notify)
   Widget widget;
   XmString item;
   Boolean notify;
```

In the Common Widgets subsystem, the XmListSelectItem call has been mapped to an instance method of the class CwList:

```
selectItem: item notify: notify
```

The C type Widget, in this case, is mapped to the Smalltalk class CwList since the XmListSelectItem function applies only to list widgets. The XmList prefix has been stripped off, because such C-specific prefixing is unnecessary in Smalltalk.

Where C types have appropriate corresponding Smalltalk base classes, C types are mapped to these. For example, the C type XmString is mapped to the Smalltalk class String, and the C type Boolean is mapped to the Smalltalk class Boolean.

Common Widgets Class Hierarchy

This section describes the Common Widgets class hierarchy and provides an overview of the functionality provided by each class.

A widget is a user interface component, such as a top-level window (shell), button or list. A graphical user interface is built by creating a tree of widgets. Every widget except the topmost widget in a tree has a parent widget. In the user interface a child widget appears on top of the parent and is normally prevented from drawing outside the bounds of its parent. Each parent widget is responsible for sizing and positioning its children. The parent-child relationship defines the widget tree. The topmost widget in the tree is called a shell widget. Shell widgets are responsible for negotiating with the window manager for their position on the screen, and for the various window decorations that they display, such as a title, border, or close box.

All Common Widgets are subclasses of one of three classes listed below.

CwPrimitive	Primitive widgets have no children
CwComposite	Composite widgets can have zero or more children
CwShell	Shell widgets have exactly one child

Primitive widgets are the simplest building blocks in Common Widgets. A primitive widget is always the child of another widget. The following table provides brief descriptions of the CwPrimitive class and its subclasses. Classes in italics are abstract classes (they are never instantiated).

Class Hierarchy	Responsibility
CwWidget	Defines common behavior for all widgets.
CwBasicWidget	Defines common behavior for widgets implemented directly by the platform.
CwPrimitive	Defines common behavior for widgets that do not have children.
CwArrowButton	Displays an arrow button.
CwComboBox	Combines a list and text area to provide a prompted entry field.
CwLabel	Displays a static label that can be a string, pixmap, or icon.
CwDrawnButton	Displays a button drawn by the application.
CwPushButton	Displays a push button containing a string, pixmap, or icon.
CwToggleButton	Displays a button that has on/off state such as a radio or check button.
CwList	Displays a list of strings from which one or more can be selected.
WbScrolledList	Displays a scrollable list of strings from which one or more can be selected.
CwScrollBar	Displays a vertical or horizontal scroll bar.
CwSeparator	Displays a separator line, normally between menu items.
CwText	Displays and provides editing capability for text.
WbScrolledText	Displays and provides editing capability for multi-lined text.

Composite widgets can have zero or more child widgets. A composite widget’s children can include other composite widgets, primitive widgets or both. Different composite widgets provide various kinds of layout capabilities for arranging children. The following table briefly explains the CwComposite class and its subclasses. Classes in italics are abstract classes.

Class Hierarchy	Responsibility
CwWidget	Defines common behavior for all widgets.
CwBasicWidget	Defines common behavior for widgets implemented directly by the platform.
CwComposite	Defines common behavior for widgets that contain child widgets.

Class Hierarchy	Responsibility
CwBulletinBoard	Allows application-defined child widget positioning.
CwForm	Provides a constraint-based mechanism for laying out child widgets.
CwDrawingArea	Performs no child layout and provides an area for performing graphic operations.
CwFrame	Provides a frame around its single child widget.
WbFrame	Provides a frame around its single child widget with an adjustable frame width.
CwRowColumn	Provides a mechanism for laying out child widgets in rows or columns.
WbRadioBox	Provides a simple Radio Button group
CwScale	Displays a numeric scale with a position indicator.
CwScrolledWindow	Allows a child widget to be scrolled using scrollbars.
CwMainWindow	Provides layout management for a menu bar and optionally scrollable work area.

Shell widgets provide the protocol between the application interface and the window manager. The following table provides brief descriptions of the CwShell class and its subclasses. Classes in italics are abstract classes.

Class Hierarchy	Responsibility
CwWidget	Defines common behavior for all widgets.
CwBasicWidget	Defines common behavior for widgets implemented directly by the platform.
CwShell	Defines common behavior for the top level widgets of windows and dialogs.
CwOverrideShell	A pop-up window that bypasses window management, and normally appears over all other widgets.
CwWMShell	Defines common behavior for shell widgets that do not bypass window management.
CwTopLevelShell	Provides a normal window with standard appearance and decorations.
CwTransientShell	A pop-up dialog window that does not bypass window management.
CwDialogShell	A pop-up window used to implement modal and modeless dialog windows.

Overview of Common Widgets User Interface Concepts

All user interfaces have two directions of communication: from the application to the user, and from the user back to the application. Using Common Widgets, these two directions of communication work as follows:

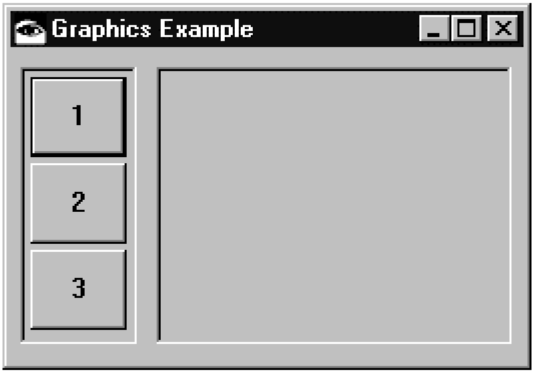
- The application creates and configures user interface components (widgets) through the use of **resources** and **functions**, and expresses interest in receiving notification of user actions by registering **callbacks** and **event handlers**.
- The user interface notifies the application of user actions by calling callbacks and event handlers.

The next sections explain how widgets are created and configured, and how callbacks and event handlers allow the application to react to user actions.

The Parent-Child Widget Tree

A widget tree is created in a top-down manner. First a shell widget is created. Next, a single child widget, usually a subclass of CwComposite, is created as the child of the shell. This process continues until the application has created all the widgets in the tree.

In the example below, a simple graphics application window is shown. A CwTopLevelShell is created to interact with the window manager. A CwMainWindow is created as the single child of the shell. A CwForm is created as the child of the main window. The CwForm is required to position a CwDrawingArea and a CwRowColumn, which are created as children of the form. Three CwPushButton’s are created as children of the row-column widget. The window looks like this:



Here is the code used to create this interface.

Example Code to Create a Widget Tree

The code below illustrates an example window built with WindowBuilder Pro.

```
setUpShell: aShell
   “Private: WARNING!!!! This method was automatically generated
    by WindowBuilder Pro. Code you add here which does not
    conform to the WindowBuilder Pro API will probably be lost
    the next time you save your layout definition.”
   aShell
      x: 100;
      y: 100;
      width: 292;
      height: 156;
      fontExtent: 7 @ 16;
      title: ‘Graphics Example’;
      mwmDecorations: MWMDECORALL;
      yourself.

addWidgets
   “Private: WARNING!!!! This method was automatically generated
    by WindowBuilder Pro. Code you add here which does not
    conform to the WindowBuilder Pro API will probably be lost
    the next time you save your layout definition.”
   | button1 button2 button3 draw rowColumn |
   rowColumn := CwRowColumn
      createWidget: ‘rowColumn’
      parent: self form
      argBlock: [:w | w
            x: 4;
            y: 4;
            width: 52;
            height: 144;
            borderWidth: 1;
            marginHeight: 5;
            marginWidth: 5;
            spacing: 5;
            scale].
   button1 := CwPushButton
      createWidget: ‘button1’
      parent: rowColumn
      argBlock: [:w | w
            x: 5;
            y: 5;
            width: 42;
            height: 40;
            labelString: ‘1’;
            scale].
```

```
button2 := CwPushButton
   createWidget: ‘button2’
   parent: rowColumn
   argBlock: [:w | w
         x: 5;
         y: 50;
         width: 42;
         height: 40;
         labelString: ‘2’;
         scale].
button3 := CwPushButton
   createWidget: ‘button3’
   parent: rowColumn
   argBlock: [:w | w
         x: 5;
         y: 95;
         width: 42;
         height: 40;
         labelString: ‘3’;
         scale].
draw := CwDrawingArea
   createWidget: ‘draw’
   parent: self form
   argBlock: [:w | w
         x: 62;
         y: 5;
         width: 222;
         height: 144;
         borderWidth: 1;
         scale].
rowColumn
   attachLeft: 4 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   yourself.
draw
   attachLeft: 6 relativeTo: rowColumn;
   attachRight: 8 relativeTo: XmATTACHFORM;
   attachTop: 5 relativeTo: XmATTACHFORM;
   attachBottom: 7 relativeTo: XmATTACHFORM;
   yourself.!
```

When this code is executed, a window titled ‘Graphics Example’ appears on the screen. The widgets behave as expected, but the application is not notified when a button is pressed or when the mouse is moved. In order for the application to be notified of the user’s interaction with the widgets, event handlers and callbacks are required.

The Widget Lifecycle

In order for a widget to appear on the user's display, it must be created, managed, mapped and realized. When a widget is no longer required, it is destroyed. These steps are described below.

1. Creating a Widget

The first step is to create the widget. When a widget is created, it is given a name, and its parent-child relationship is established. A new widget adopts default settings for any of its resources that are not explicitly set when the widget is created. Widget resources are data that define how a widget appears and behaves. Resources are described in the next section.

Widget names serve several purposes:

- Some widgets use their name to establish a default resource value. For example, shells that have titles use their name as the default title. Push button and label widgets use their name as the default labelString. Using the name in this way results in more efficient code.
- Widget names are used as part of a widget's printable representation (using printOn:). They are also useful for identifying widgets during debugging.
- On platforms supporting OSF/Motif as the native widget system, widget names are used in conjunction with X resource files to set initial resource values for widgets.

2. Managing a Widget

Managing a widget specifies that its size and position will be managed by its parent widget. This process is called **geometry management**. Any changes in the size or position of the parent widget will be recursively propagated to managed child widgets. The application must specify that a widget is to be managed by sending the manageChild message to the widget.

Managing or unmanaging a widget does not affect the managed state of child widgets. However, if a widget is unmanaged then neither it nor any of its children will participate in geometry management. By default a widget is not managed when it is created.

3. Mapping a Widget

Mapping a widget specifies that it is to appear on the display once it has been **realized**. Realizing a widget is described below. By default, a widget is mapped automatically when it is managed. This is controlled by the setting of the `#mappedWhenManaged` widget resource, which defaults to true. Unmanaging a widget also unmaps it.

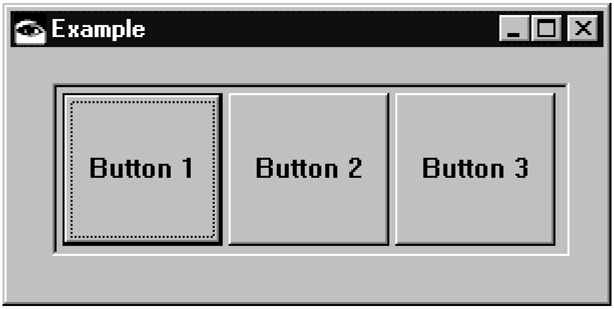
It is possible for widgets to be created, managed, and realized, but left unmapped. The widget remains invisible until it is mapped. This technique is useful for quickly displaying frequently used dialog boxes.

Widgets are mapped using the `#mapWidget` method, and unmapped using the `#unmapWidget` method. Mapping or unmapping a widget maps or unmaps all child widgets.

4. Realizing a Widget

Up until the time a widget is realized, it is invisible. It can be created, managed in the tree, and mapped, but it will not appear on the screen until it is realized. During realization, all widgets assume their initial geometry and create their visual appearance. Widgets are realized by sending the `#realizeWidget` message to the topmost widget in the hierarchy, or shell widget. Realizing a widget realizes all of its children recursively. Widgets created, mapped, and managed as children of already *realized* widgets are automatically realized on creation.

In the example below, the CwTopLevelShell widget has been realized, and the CwRowColumn has positioned its three CwPushButton children.



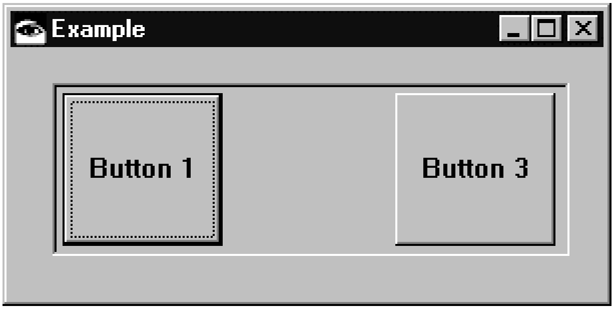
5. Destroying a Widget

When a widget is no longer required, it is destroyed. A widget can be implicitly destroyed by the user, for example by clicking on a close box. Alternatively, a widget can be destroyed under application control by sending it the `#destroyWidget` message. The widget is removed from the display and released from memory. Destroying a widget recursively destroys all child widgets.

Mapping and Unmapping Widgets

A widget that is managed but not mapped still participates in geometry management, that is, it takes up space in its parent, but it is not actually drawn. Normally this results in a blank area in the parent where the widget would otherwise appear.

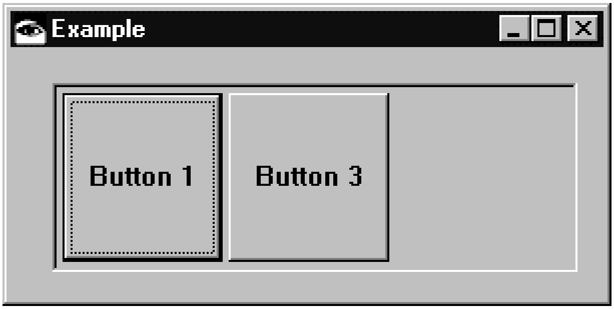
In this example, assume that the widget tree is created, managed and realized, but the second CwPushButton is subsequently unmapped. Notice that the button is removed from the screen, but its parent still reserves its position in the row-column widget.



Managing and Unmanaging Widgets

When a widget is unmanaged, its parent can reclaim the space the widget occupies. If the parent widget is a composite that performs layout of its children, it will adjust child layout accordingly. A row-column was chosen for this example because it provides a visual demonstration of the difference between mapping and managing widgets.

In this example, assume that the widget tree is created, managed and realized, but CwPushButton 2 is subsequently unmanaged. Not only is the button removed from the screen, or unmapped, but it also loses its position in the row-column widget.



Widget Resources and Functions

Widgets are configured and controlled by the application through resources and functions. Widget **resources** are named stated that define the behavior and appearance of a widget. Widget **functions** are messages that can be sent to a widget to tell it to do something.

In VisualAge, resource accessors and functions are implemented as Smalltalk methods in widget classes. Widget **resource methods** provide access to a widget's resources. All methods that are not resource methods are widget **function methods**. A method's comments and its message specification indicate whether it is a resource method or a function method.

Resources

Resources are somewhat analogous to Smalltalk instance variables, and resource set and get methods are similar to instance variable accessor methods. However, there are several important differences:

- Resources may or may not be implemented using instance variables, and the implementation varies from platform to platform.
- Changes to values in widget resources are immediately reflected in the appearance of the widget.
- On platforms running Motif as the native widget system, VisualAge widget resource values can be set using standard X resource files, as with any other Motif application.

All widgets have a core set of resources. For example, all widgets have width and height resources. A widget can also have resources specific to its behavior. For example, the items resource of the CwList widget defines what items are displayed in the widget. Default values are provided for all of a widget's resources. A widget's resources are set or retrieved using the set and get accessor methods of the widget, the names of which correspond directly to the associated resource name.

Resources defined by a widget's superclasses are inherited. For example, consider the widget class CwPushButton. This class is a subclass of CwLabel, CwPrimitive, CwBasicWidget and CwWidget. CwPushButton inherits resources from all of its superclasses, as well as defining additional resources of its own.

The following table illustrates the resources that are available for a CwPushButton widget. Many of the resources are provided by CwWidget, and these are available to all widgets.

CwWidget	CwPrimitive	CwLabel	CwPushButton
ancestorSensitive	backgroundColor	accelerator	activateCallback
borderWidth	foregroundColor	acceleratorText	armCallback
depth	helpCallback	alignment	disarmCallback
destroyCallback	navigationType	fontList	showAsDefault
height	traversalOn	labelIcon	
mappedWhenManaged		labelInsensitiveIcon	
resizable		labelInsensitivePixmap	
resizeCallback		labelPixmap	
sensitive		labelString	
userData		labelType	
width		marginBottom	
x		marginHeight	
y		marginLeft	
(plus 16 attachment-related		marginRight	
resources)		marginTop	
		marginWidth	
		mnemonic	
		recomputeSize	

Not all widget resources can be modified or retrieved at any time. Resources are tagged with the letters C, S, or G to indicate when the resource can be modified or retrieved, as follows:

- The application can set the resource at creation time only (C).

- The application can set the resource at any time (S).
- The application can retrieve, or get, the resource at any time *after* the widget is created (G).

The specification for each resource method provides the resource access designation information. Most resources are designated with all three attributes (C, S, and G). Resources are manipulated using get and set accessor methods derived from the OSF/Motif name for the resource by removing the ‘XmN’ prefix. For example, the Motif resource *XmNheight* for a widget is retrieved and modified using the `#height` and `#height`: methods, respectively.

Create-only (C) resources can only be set using an *argBlock* at widget creation time. An *argBlock* is a single argument Smalltalk block of code that is evaluated with the widget being created as its argument. Resources with an (S) designation can also be set in the *argBlock*. The *argBlock* is evaluated before the widget is fully created, that is, while it is still under construction, but after a Smalltalk object has been created to represent the widget. If no *argBlock* is required, nil can be used for the *argBlock* argument, rather than unnecessarily creating an empty block.

Tip: Always set resources in the create *argBlock* wherever possible. If the system has more information available at the time of widget creation, it can perform more optimization. On some platforms a significant performance advantage is achieved by setting resources in the create *argBlock* rather than immediately after creation, which may cause default widget configuration to have to be “undone”.

In the following example, the width and height resources for a drawing area are explicitly set in an *argBlock* when the drawing area widget is created. These specify the size in pixels of the drawing area widget. The size of the shell widget is calculated based on the size of the drawing area widget. In general, when the size of a widget is not explicitly specified, it is calculated based on the size of its children, recursively. The string arguments in the creation messages specify the names of the widgets. By default, the name of the top level shell appears as the window title.

```
| shell drawingArea |
shell := CwTopLevelShell
   createApplicationShell: ‘ShellName’
   argBlock: nil.
drawingArea := CwDrawingArea
   createWidget: ‘draw’
   parent: shell
   argBlock: [:w | w
      width: 100;
      height: 100].
drawingArea manageChild.
shell realizeWidget
```

Resources with set (S) and get (G) designations can be set and retrieved, respectively, after widget creation using the appropriate set and get methods.

Multiple resources with set (S) designation can also be set simultaneously after the widget has been created using the `#setValuesBlock:` message, which takes an *argBlock* as argument. The `#setValuesBlock:` method is the recommended way of setting multiple resources for a widget after the widget is created. Normally, after a widget has been created and a resource is modified that changes a widget's appearance, the widget is redisplayed to show the change. Using a `#setValuesBlock:` is more efficient than setting the resources outside the block because the widget can then optimize updates together, even if several of them change the widget's appearance. The block passed to `#setValuesBlock:` has the same format as the *argBlock* used when creating a widget.

In the following example, the geometry of a shell widget is changed. Assume that the variable shell is a top level shell that has been created and realized.

```
“Set widget geometry using a series of set accessor methods. The
shell is redrawn up to four times, once for each resource change.”
shell
   x: 10;
   y: 10;
   width: 100;
   height: 100.
“Set widget geometry using a set values block. The shell is
redrawn only once, with the final dimensions, at the final
position.”
shell
   setValuesBlock: [:w | w
      x: 10;
      y: 10;
      width: 100;
      height: 100].
```

Tip: Some resources change their values when the value of a different resource in the same widget is changed. To avoid this “push-down-here-pop-up-there” effect, such resources must be set simultaneously using an *argBlock*, either on creation or after creation using `#setValuesBlock:`. This situation occurs with left/right and top/bottom CwForm attachment resources, which should always be set in pairs.

Function Methods

Widget methods that are not resource set or get methods are widget **function methods**. Unlike resource setting messages, function messages can only be sent to widgets after they have been created. While resource methods are used to access or change widget state, function methods typically perform more complex operations, and in some cases modify resource values. While resource get and set methods uniformly require zero arguments and one argument respectively, widget function methods take varying numbers of arguments, depending on the particular function. The `#manageChild` method is an example of a widget function.

Functions often alter the resource values of a widget as a side effect. For example, the `#setString:` function for text widgets alters the value resource of the widget. In some cases it is possible to achieve the same effect by using either a resource method or a function method.

Tip: Do not call function methods from inside a create *argBlock*. Because the widget is not fully created when the create *argBlock* is evaluated, invoking widget functions will result in errors.

CwConstants Pool Dictionary

The Common Widgets subsystem uses a pool dictionary called CwConstants to provide pool variables for constant values. For example, pool variables such as XmATTACHFORM and XmNactivateCallback are used as arguments to Common Widgets methods. These pool variable names should be used rather than directly using their constant values. All classes that require these Common Widgets pool variable names must include the CwConstants pool dictionary in their class definition.

Widget Event Handling and Callbacks

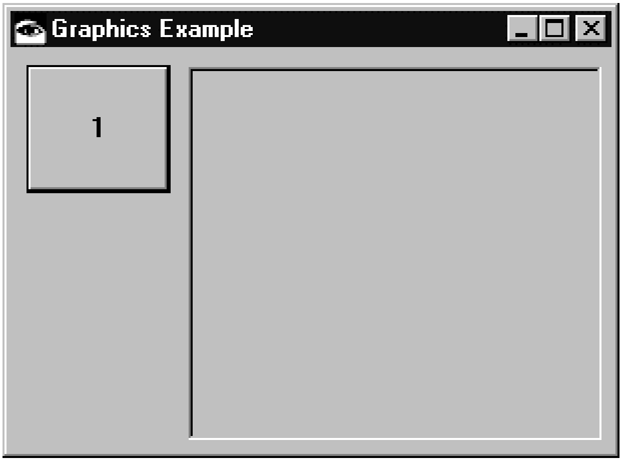
An **event** is the mechanism that notifies the application when the user performs a mouse or keyboard operation. The application can be notified about key presses and releases, mouse button presses and releases, and mouse movements. Events are handled by adding an event handler to a widget.

A **callback** is the mechanism that notifies the application when some higher level action is performed on a widget. For example, the *XmNactivateCallback* is used to inform the application that a CwPushButton has been pressed and released. As another example, all widgets support the *XmNdestroyCallback* that is invoked just before a widget is destroyed.

The following example illustrates how callbacks and event handlers are defined.

Example of Using an Event Handler and a Callback

In the example below, a small graphics application interface is created. The window created by the code is illustrated below:



The code to create this window is similar to the earlier example, but in this example event and callback handler code (**bold text**) has been added. This code registers the event and callback handlers just after the widgets are created.

When the push button widget is pressed (that is, when the user clicks mouse button 1 while the mouse pointer is over the push button widget), the `#pressed:clientData:callData:` method is executed.

When the mouse is moved in the drawing area with button 1 held down, the `#button1Move:clientData:event:` method is executed.

```
setUpShell: aShell
   “Private: WARNING!!!! This method was automatically generated
    by WindowBuilder Pro. Code you add here which does not
    conform to the WindowBuilder Pro API will probably be lost
    the next time you save your layout definition.”
   aShell
      x: 100;
      y: 100;
      width: 284;
      height: 168;
      fontExtent: 7 @ 16;
      title: ‘Graphics Example’;
      mwmDecorations: MWMDECORALL;
      yourself.

addWidgets
   “Private: WARNING!!!! This method was automatically generated
    by WindowBuilder Pro. Code you add here which does not
    conform to the WindowBuilder Pro API will probably be lost
    the next time you save your layout definition.”
```

```
| button draw |
button := CwPushButton
   createWidget: ‘button’
   parent: self form
   argBlock: [:w | w
          x: 4;
          y: 4;
          width: 56;
          height: 56;
          labelString: ‘1’;
          scale].

draw := CwDrawingArea
   createWidget: ‘draw’
   parent: self form
   argBlock: [:w | w
          x: 64;
          y: 4;
          width: 212;
          height: 156;
          borderWidth: 1;
          scale].

button
   attachLeft: 4 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   addCallback: XmNactivateCallback
          receiver: self
          selector: #pressed:clientData:callData:
          clientData: nil;
   yourself.

draw
   attachLeft: 4 relativeTo: button;
   attachRight: 8 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   addEventHandler: Button1MotionMask
          receiver: self
          selector: #button1Move:clientData:callData:
          clientData: nil;
   yourself.
```

Fonts

The font used by certain widgets can be specified by the application. The following widgets allow their font to be changed: CwLabel, CwPushButton, CwToggleButton, CwText, CwList, CwComboBox, and CwScale. The font is changed using the `#fontList:` method. The font to use is specified by a CwFontList object.

To create a CwFontList, the `#fontStruct:` class method of CwFontList is passed a CgFontStruct describing a Common Graphics font. A CgFontStruct can be loaded using the `#loadQueryFont:` method of CgDisplay. For further details on fonts, consult “Using Fonts” in the *VisualAge Programmer’s Reference*.

The following code creates a multi-line text widget and sets its font to the monospaced font named ‘8x13’.

```
| shell fontStruct fontList text |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Font List Example’].
fontStruct := shell display loadQueryFont: ‘8x13’.
fontList := CwFontList fontStruct: fontStruct.
text := shell
   createText: ‘text’
   argBlock: [:w | w
      editMode: XmMULTILINEEDIT;
      fontList: fontList].
text setString: ‘This text is displayed using the 8x13 font.’.
text manageChild.
shell realizeWidget
```

Using the System Browser Font

All of the browsers in VisualAge are subclasses of EtWindow. This class keeps one font that every browser uses. You can find the browser font name by evaluating:

```
   EtWindow fontName.
```

You can change the browser font from the File menu. If the browser font has not been changed, then the EtWindow class method fontName returns nil. If your window will use the browser font, then you can make the window a subclass of EtWindow. Your subclass should provide the instance method `#fontSettableWidgets`, which answers a collection of all the widgets to be notified in case the font changes. EtWindow calls all of these widgets for you and tells them to change to the new font.

You can still use the browser font, even if your window does not subclass EtWindow. The example below creates a new window with the system font. The class method fontList in EtWindow returns either the current CwFontList, or nil if the font has not been changed.

```
|shell text fontList|
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Browser Font’].
fontList := EtWindow fontList.
fontList isNil ifTrue: [
   fontList := CwFontList fontStruct:
      (CgDisplay default defaultFontStruct)].
text := shell
   createText: ‘text’
   argBlock: [:w | w
      columns: 60;
      editMode: XmMULTILINEEDIT;
      fontList: fontList].
text setString: ‘This font is the system browser font.’.
text manageChild.
shell realizeWidget
```

Colors

The background and foreground color of widgets can be set and queried by the application using the backgroundColor and foregroundColor resources. The foregroundColor is used for text or other foreground graphics, and the backgroundColor is used to fill the background of the widget. The color values are specified using CgRGBColor objects which allow the application to specify the red, green, and blue components of the desired color. See “Specifying Colors” in the *VisualAge Programmer’s Reference* for more information concerning the use of CgRGBColor objects. There are platform-specific limitations concerning setting the colors of certain widgets. See Chapter 12, “Widget Encyclopedia,” for the details of these limitations.

Note: Due to platform-specific limitations, a widget may not take on a requested color setting, or it may take on a slightly different color setting than requested. To determine the exact color a widget is using, the resource can be queried after it is set. Querying the color resource always returns the color the widget is actually using.

The following code creates a multi-line text widget and sets its foregroundColor to black and its backgroundColor to red.

```
| shell text |
shell := CwTopLevelShell
  createApplicationShell: ‘shell’
  argBlock: [:w | w title: ‘Color Example’].
text := shell
  createText: ‘text’
  argBlock: [:w | w
    editMode: XmMULTILINEEDIT;
    foregroundColor: (CgRGBColor red: 0 green: 0 blue: 0);
    backgroundColor: (CgRGBColor red: 65535 green: 0 blue: 0)].
```

```
text manageChild.
```

Drag Drop Support

CommonWidgets provides no support for direct manipulation (a.k.a. “drag and drop”). EwDragAndDropSupport provides pluggable drag and drop on the base widgets without modifying them.

The Players

The Application

An application may wish to support drag and drop for a variety of reasons. It is not the purpose of EwDragAndDropSupport to prescribe the semantics of drag and drop; this is left to the application. The primary role of the application in drag and drop is to create the widgets and adapters required, hook the drag and drop callbacks on the adapters, and then make the changes to the business model as drag and drop occurs.

The Widgets

Since the base common widgets do not provide any support for drag and drop, the widgets only requirement in supporting drag and drop is that they provide mouse and keyboard events. The adapters then hook these events to detect and play out a drag and drop operation.

The Adapters

EwDragAndDropSupport provides two kinds of adapters for use in drag and drop: source adapters and target adapters.

The application instantiates a source adapter for each widget which the application designates as a drag and drop source. The source adapter serves as a wrapper around the source widget, providing a drag and drop callbacks which base widgets do not provide.

The application creates one source adapter for each source widget as follows:

```
aSourceAdapter := EwSourceAdapter on: aWidget.
```

The source adapter provides the following callbacks:

XmNdragStartCallback
XmNdragChangeCallback
XmNdragCompleteCallback
XmNdragCancelCallback

The application also instantiates a target adapter around each widget which is to be a target for drag and drop. The target adapter is created as follows:

```
aTargetAdapter := EwTargetAdapter on: aWidget.
```

The target adapter provides the following callbacks:

XmNdragOverCallback
XmNdragLeaveCallback
XmNdropCallback
XmNdragCancelCallback

The Drag Drop Manager

A central drag and drop manager instance stores system-wide drag and drop parameters. These include the list of possible operations, the cursors which correspond to each operation, and the mappings from keyboard combinations to operations. The class EwDragAndDropManager provides API to set and query these settings.

Sequence of Events

Setup

The application sets up for drag and drop by creating a source adapter on the widgets which are to act as the drag source. The application then hooks the source adapter's XmNdragStart and XmNdragComplete callbacks and optionally the XmNdragChange and XmNdragCancel callbacks.

The application also creates a target adapter on each widget which can act as a drop target, including any source widgets which can also act as targets. The application then hooks each target adapter's XmNdragOver and XmNdrop callbacks and optionally the XmNdragLeave and XmNdragcancel callbacks.

Drag Initiation

The user initiates a drag by pressing the drag mouse button over a drag source widget and moving the mouse a certain nominal distance. The source adapter, having hooked the mouse events on the source widget, detects that a drag is starting and fires the XmNdragStart callback to the application. The calldata for the XmNdragStartCallback includes the actual mouse event which triggered the drag start, the items in the source widget which are being dragged along with each item's offset from the mouse location. It also contains a default image for each item to be used to represent the item during the drag. The images and offsets may be altered by the application to cause alternate images and offsets to be used. For example, an application may wish to use a multi-file icon to represent all items being dragged rather than a single icon for each item.

The images and offsets which the adapter provides as defaults depend heavily on the API of the source widget. For example, since CwList provides no API to allow the source adapter to determine which item in the list is under the cursor when the drag starts, the adapter instead simply provides the selected items as the drag items. Similarly, since the source adapter cannot determine the offsets of the selected items, it provides offsets which cause the drag items to be beveled up and to the right from the mouse.

The XmNdragStart calldata also contains a *doit* flag which is set to true by default. The application can change this flag to false if it determines that dragging is not allowed for some reason. Finally, the calldata includes a default vote, which is an array of operations which the source will allow for these items. The application can change this vote to an Array of operations which it will allow, given the items being dragged.

Dragging Over a Target

Each time the mouse moves, the system determines which widget is now under the mouse. The system keeps a registry of all target adapters and their widgets. This enables it to map the widget under the mouse (if any) to its corresponding target adapter (if any).

If no widget is under the mouse or if no target adapter exists for a widget, the no parking cursor is automatically shown. If a target adapter is found, however, it fires its XmNdragOver callback to its application. The XmNdragOver calldata contains the items being dragged and the source widget as well as the mouse event. The target adapter also determines which item in the target widget is under the mouse and supplies this in the calldata. This is only possible on widgets that provide the necessary API to determine this. Since no base widgets provide API to support this, the item under the cursor is always nil for target adapters on base widgets.

If there is an item under the cursor (as is the case in some extended widgets), the application must determine whether the item itself is capable of accepting a drop. For example, a trash can or a printer typically would be able to accept a drop, while an employee object typically would not.

The calldata also contains a flag by which the application can define which kind of emphasis is to be shown on the target widget. Again, this is not possible on any of the base widgets, only on some extended widgets. If the application has determined that the item under the cursor is in fact capable of receiving the drop, it sets the emphasis flag to XmTARGETEMPHASIS. If it determines that the item is not capable of receiving the drop, then it can set the emphasis flag to either XmINSERTIONEMPHASIS or XmNOEMPHASIS (default). XmINSERTIONEMPHASIS indicates that if the drag items are dropped, they will be inserted into the target widget at the index determined by the mouse position.

Finally, the XmNdragOver calldata contains a vote. The application can set this to be an array of operations which it will allow given the current target widget, target item (if any), and the items being dragged.

In some cases, an application may need to draw while a drag is in progress. To avoid leaving visual "debris" on the screen, the application must do any drawing from within a call to #drawDuringDrag:, which takes a Block as its argument. All drawing which takes place in the block is guaranteed not to leave debris on the screen.

Voting and Cursors

As items are being dragged, the user can affect the voting by pressing the SHIFT and/or CTRL keys. The meaning of each key combination is configurable in the EwDragAndDropManager. Each time a dragOver callback fires and each time the keyboard status changes, the source, target, and keyboard votes are recalculated, and a net operation is determined. This operation, in turn, determines what the cursor should be. The cursor to be used for each operation is also configurable in the EwDragAndDropManager.

Since dragging images causes the cursor to blink, the EwDragAndDropManager also provides and alternative to cursors called "cursor images." Like cursors, each cursor image corresponds to an operation. The difference is that if a cursor image is defined for an operation, it is drawn over all the drag images and will not blink as the mouse is moved. By default, the system turns off the cursor during drag and drop and uses a unique cursor image for each operation. These can all be changed and customized via API in the EwDragAndDropManager.

Source Vote and Image Changes

Some applications may require that the source be able to change its vote, its drag images, or its drag offsets based on the target and/or the operation. For example, if the source widget is a list of object templates, the source may want to change the icon from the template icon to an instance icon once the item is dragged outside the source widget. As another example, the source may want to allow certain operations only if the item is being dragged over the source. In this case, the source would need an opportunity to change its vote whenever the target changed.

To support this requirement, the XmNdragChange callback on the source adapter is fired whenever the target or the operation changes. The calldata includes the items being dragged, the target widget, and the current operation. It also includes the drag images and offsets as well as the most recent source vote. The application can change these values to implement behaviors like those outlined in the examples above.

Leaving a Target

In some cases, an application may need to be notified when items are dragged away from a target. For example, if an application had hooked the XmNdragOver callback to change the trash can's icon to show the lid up, it would need to be notified when the drag had left that target widget so that it could change the icon to show the lid back down.

To support this, the XmNdragLeave callback on the target adapter is fired whenever the items are dragged away from the target of the last XmNdragOver callback. The calldata includes only the source widget and the source items. Neither of these fields may be changed; this callback is for notification only.

Dropping

When the user releases the mouse button, the drop occurs on the target widget. Both the target and the source need to be notified. The target adapter first fires the XmNdrop callback. The calldata includes the source widget, the source items, the operation, and the mouse event. It also includes an offset for each source item relative to the mouse location. If, in response to the last XmNdragOver callback, the application set the emphasis to XmTARGETEMPHASIS, the calldata for the XmNdrop callback also contains the target item. If the emphasis was set to XmINSERTIONEMPHASIS, the calldata contains the insertionIndex. Finally, the calldata contains a doit flag, which the application can set to false if it is unable to perform the drop. In this case the source adapter will fire its XmNdragCancel callback. Otherwise, it is the application's responsibility to perform the appropriate operation on the target widget and the business model.

After the target has fired the XmNdrop callback, the source adapter fires its XmNdragComplete callback. The calldata includes the source items, the target widget, the target item (if any), and the operation. It is the application's responsibility to perform the appropriate operation on the source widget and the business model. For example, if the operation was XmMOVE, the application should remove the items from the source widget.

Canceling a Drag

The user may cancel a drag at any time by pressing XkCancel. In this case the source adapter and the target adapter (if any) fire their XmNDragCancel callback. The calldata for this callback depends on whether the adapter is a source or a target adapter. For source adapters, the calldata includes the same information as the calldata for the XmNdragComplete callback. For target adapters, the calldata includes the same information as the calldata for the XmNdrop callback.

The source adapter also fires the XmNdragCancel callback when the items are dropped outside any target widgets. The XmNdragCancel callback is also fired when the items are dropped onto a target that is unwilling to accept them, that is, when no valid operation is in effect or when the target widget's application sets the doit flag to false in the XmNdropCallback's calldata.

System Configuration

A number of aspects of drag and drop are configurable via API on the EwDragAndDropManager.

The set of operations and their relative priorities are configurable. By default, the operations are XmMOVE, XmCOPY, and XmMIRROR. (Mirror means making the same item be in more than one widget.) When the source, target, and keyboard votes are tallied, the highest priority operation of the intersection of the three votes is used as the operation. The priority of the votes is determined by the order in which the source answers its allowable operations in the dragStart callback. The highest priority vote is the first vote the source gives. If the intersection is empty, then the no parking cursor is shown.

The mapping of the SHIFT and CTRL key combinations can be customized also. The default mappings are:

None: all operations are allowable
Shift: XmMOVE
Control: XmCOPY
Both: XmMIRROR

The drag and drop manager also decides which cursor to use based on the operation. As mentioned above, since dragging images causes the cursor to blink, the EwDragAndDropManager also provides and alternative to cursors called “cursor images.” Like cursors, each cursor image corresponds to an operation. The difference is that if a cursor image is defined for an operation, it is drawn over all the drag images and will not blink as the mouse is moved. By default, the system turns off the cursor during drag and drop and uses a unique cursor image for each operation.

Minimal Drag Drop

For most applications, the default settings are acceptable. This section describes what the minimum requirements are for an application to enable drag and drop.

At the minimum, the application must create the source and target adapters on the appropriate widgets. On the source adapter it does not need to hook the XmNdragStart callback unless it intends to change the vote, images, or offsets to values other than the defaults or unless it wants to deny drag and drop in certain cases by setting the doit flag false. The application will have to hook the XmNdragComplete callback to perform the operation. The XmNdragChange callback does not need to be hooked, nor does the XmNdragCancel callback.

On the target adapters, the application will only need to hook the XmNdragOver callback if it needs to check the kind of items being dragged to ensure that the target widget can receive them or if it does not want the default target vote. It will have to hook the XmNdrop callback to perform the operation. It does not need to hook the XmNdragLeave or XmNdragCancel callbacks.

Drag Drop on Base Widgets

Technically, any base widget that provides mouse and keyboard events can be wrapped in a source adapter, and any widget can be wrapped in a target adapter. Since there are no commonly accepted semantics for drag and drop on labels, buttons, forms, or other non-list-oriented widgets, the application must define what is meant, for example, by dragging from a label to a toggle button.

Also, since most base widgets are not item-oriented, the target callbacks do not provide any item under cursor information, and the source callbacks do not provide any source item information.

Drag and drop is best suited for use on container style extended widgets, as provided by EwContainerSupport.

Chapter 10 Callbacks and Event Handlers

Callbacks

Actions performed on widgets by the user must be communicated back to the application. One mechanism used for this communication is a *callback*. A **callback method** defines actions to perform in response to some occurrence in a widget. Callbacks are normally registered just after widgets are created. For example, when a push button widget is created, the application usually registers an **activate** callback that is executed when the button is activated by the user clicking on it. Although it is not necessary for the application to register callbacks, without them the application is unable to take action based on the user's interaction with the widgets.

Callbacks are registered using the `#addCallback:receiver:selector:clientData:` method.

Tip: The *argBlock* argument of a widget creation message can only be used to set widget resources. The `#addCallback:...` message cannot be used within the create *argBlock*. Callbacks are usually registered immediately after the widget has been created, and before it is realized.

`The #addCallback:receiver:selector:clientData:` method takes 4 arguments:

callbackName A constant specifying which callback is being registered, for example, XmNactivateCallback

receiver The object that will receive the callback message

selector The 3-argument callback message selector (WindowBuilder Pro also supports unary callback message selectors)

clientData Optional data that will be passed to the callback when it is executed

When a callback method is executed, it is passed three arguments:

- The widget that caused the callback
- The client data specified when the callback was registered
- Information specific to the type of callback, called the call data

The example below illustrates how to register a callback. First a button is created, in this case, as the child of a shell, and then an XmNactivateCallback is added to the button.

```
| shell button |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: nil.
button := shell
   createPushButton: ‘OK’
   argBlock: nil.
button
   addCallback: XmNactivateCallback
   receiver: self
   selector: #pressed:clientData:callData:
   clientData: ‘Test data’.
button manageChild.
shell realizeWidget.
```

When an activate callback occurs due to the button being pressed, the `#pressed:clientData:callData:` method, shown below, is executed. The method prints the string ‘Test data’ in the Transcript window. The widget issuing the callback is passed as the widget argument. In this case, this is the push button widget. The string ‘Test data’, specified as the client data when the callback was added, is passed as the clientData argument. The callback-specific data is passed as the callData argument. For the activate callback of push button widgets, however, the call data provides no new information.

```
pressed: widget clientData: clientData callData: callData
   “The push button has been pressed.”
   Transcript cr; show: clientData
```

The following table describes the class hierarchy and data accessor method names for call data objects. All classes are concrete classes.

Class Hierarchy	Responsibility	Data Accessor Methods
CwAnyCallbackData	Provides call data for most callbacks.	reason (a constant, prefixed by `XmCR’)
CwComboBoxCallbackData	Provides call data for combo box	singleSelectionCallback. item itemPosition
CwConfirmationCallbackData	Provides call data for vendor shell windowCloseCallback. This callback can be cancelled by the application.	doit doit:
CwTextVerifyCallbackData	Provides call data for text and combo box modifyVerifyCallback. These callbacks can be cancelled by the application.	currInsert endPos startPos text text:
CwDrawingCallbackData	Provides call data for drawing area input, expose and interceptExpose callbacks, and drawn button activate and expose callbacks	event window
CwListCallbackData	Provides call data for list browseSelect, singleSelect, multipleSelect, extendedSelect, and defaultAction callbacks.	item itemPosition selectedItemCount selectedItemPositions selectedItems
CwRowColumnCallbackData	Provides call data for row-column entryCallback.	widget data callbackData
CwToggleButtonCallbackData	Provides call data for toggle button valueChangedCallback.	set
CwValueCallbackData	Provides call data for scale and scroll bar valueChangedCallback, and scroll bar decrement, increment, pageDecrement, pageIncrement, toBottom, and toTop callbacks.	value

The following table lists the callbacks supported by each widget.

Widgets	Callbacks Supported
CwArrowButton	activate, arm, disarm, destroy, help, resize
CwBasicWidget	destroy, resize
CwCascadeButton	cascading, destroy, help, resize
CwComboBox	activate, destroy, focus, help, losingFocus, modifyVerify, resize, singleSelection, valueChanged
CwComposite	destroy, expose, interceptExpose, resize
CwDialogShell	destroy, focus, iconify, popdown, popup, resize, windowClose
CwDrawingArea	destroy, expose, focus, help, input, interceptExpose, losingFocus, resize
CwDrawnButton	activate, arm, destroy, disarm, expose, focus, help, losingFocus, resize
CwForm	destroy, expose, focus, help, interceptExpose, losingFocus, map, resize, unmap
CwFrame	destroy, expose, focus, help, interceptExpose, losingFocus, resize
CwLabel	destroy, help, resize
CwList	browseSelection, defaultAction, destroy, extendedSelection, help, multipleSelection, resize, singleSelection
CwMainWindow	destroy, expose, focus, help, interceptExpose, losingFocus, resize
CwOverrideShell	destroy, popdown, popup, resize
CwPrimitive	destroy, help, resize
CwPushButton	activate, arm, destroy, disarm, help, resize
CwRowColumn	destroy, entry, expose, focus, help, interceptExpose, losingFocus, map, resize, simple, unmap
CwScale	destroy, drag, expose, focus, help, interceptExpose, losingFocus, resize, valueChanged
CwScrollBar	decrement, destroy, drag, help, increment, pageDecrement, pageIncrement, resize, toBottom, toTop, valueChanged
CwScrolledWindow	destroy, expose, focus, help, interceptExpose, losingFocus, resize
CwSelectionBox	apply, cancel, destroy, expose, focus, help, interceptExpose, losingFocus, map, noMatch, ok, resize, unmap
CwSeparator	destroy, help, resize
CwShell	destroy, popdown, popup, resize
CwText	activate, destroy, help, focus, losingFocus, modifyVerify, resize, valueChanged
CwToggleButton	arm, destroy, disarm, help, resize, valueChanged
CwTopLevelShell	destroy, focus, iconify, popdown, popup, resize, windowClose

Widgets	Callbacks Supported
CwWidget	destroy, dragDetect, resize
CwWMShell	destroy, focus, iconify, popdown, popup, resize, windowClose
WbFrame	destroy, expose, focus, help, interceptExpose, losingFocus, resize
WbRadioBox	destroy, entry, expose, focus, help, interceptExpose, losingFocus, map, resize, simple, unmap
WbScrolledList	browseSelection, defaultAction, destroy, extendedSelection, help, multipleSelection, resize, singleSelection
WbScrolledText	activate, destroy, help, focus, losingFocus, modifyVerify, resize, valueChanged

Event Handlers

Event handlers are another mechanism used to inform the application of input actions by the user. While callbacks notify the application of high level interactions such as the selection of items in a list widget, event handlers notify the application of low level interactions, including the following:

- Mouse pointer motion
- Mouse button presses and releases
- Individual key presses and releases

Event handlers are registered using the `#addEventHandler:receiver:selector:clientData:` method.

Tip: The *argBlock* argument of a widget-creation message can only be used to set widget resources. The `#addEventHandler:...` message cannot be used within the create *argBlock*. Event handlers are usually registered immediately after the widget has been created, and before it is realized.

The `#addEventHandler:receiver:selector:clientData:` method takes 4 arguments:

eventMask a bit mask specifying which events to notify the receiver of

receiver the object that is to receive the event handler message

selector the 3-argument event handler message selector (WindowBuilder Pro also supports unary event handler message selectors)

clientData optional data that is passed to the event handler when it is executed

The eventMask is specified as the logical-or of one or more of the bit masks described in the following table.

Event Masks	Description
KeyPressMask	Keyboard key down events
KeyReleaseMask	Keyboard key up events
ButtonPressMask	Mouse button down events
ButtonReleaseMask	Mouse button up events
PointerMotionMask	All pointer motion events
Button1MotionMask	Pointer motion events while button 1 is down
Button2MotionMask	Pointer motion events while button 2 is down
Button3MotionMask	Pointer motion events while button 3 is down
ButtonMotionMask	Pointer motion events while any button is down
ButtonMenuMask	Button menu request events

When an event handler method is executed, it is passed three arguments:

- The widget to which the handler was added and in which the event occurred
- The client data specified when the event handler was registered
- An object describing the event, called the event

The following table describes the class hierarchy for event objects. Classes in italics are abstract classes.

Class Hierarchy	Responsibility
CwEvent	Defines common behavior for event data in event handlers.
CwExposeEvent	Provides event data for expose events in expose callbacks (see Note below).
CwInputEvent	Defines common behavior for button, key, and motion event objects.
CwButtonEvent	Provides event data for mouse button press/release events.
CwKeyEvent	Provides event data for key press/release events.
CwMotionEvent	Provides event data for mouse motion events.

Note: An expose event handler cannot be explicitly added to a widget. A CwExposeEvent object is passed to an application as part of the call data for an *exposeCallback*.

The following messages can be sent to the event object to retrieve information about the event.

For all events (CwEvent):

type The type of event that occurred. This has one of the following values: ButtonPress, ButtonRelease, Expose, KeyPress, KeyRelease, MotionNotify.

window The CgWindow associated with the widget for which the event was generated.

display The CgDisplay associated with the event.

For expose events (CwExposeEvent):

count The number of expose events which remain for the affected CgWindow. A simple application might want to ignore all expose events with a nonzero count, and perform a full redisplay if the count is zero.

rectangle A rectangle describing the damaged area, in the coordinate system of the affected CgWindow.

x, y The x and y coordinates of the origin of the damaged rectangle.

height, width The height and width, in pixels, of the damaged rectangle.

For input events (CwButtonEvent, CwKeyEvent, and CwMotionEvent):

state A bit mask representing the logical state of modifier keys and pointer buttons just prior to the event. Possible bit masks include: *ControlMask*, *ShiftMask*, *LockMask*, *Mod1Mask* to *Mod5Mask*, and *Button1Mask* to *Button3Mask*.

x, y The x and y coordinates of the pointer, relative to the widget in which the event occurred.

point x @ y

xRoot, yRoot The coordinates of the pointer, relative to the screen.

pointRoot xRoot @ yRoot

time The time, in milliseconds, at which the event occurred.

For mouse button events (CwButtonEvent):

button The number of the button that was pressed or released (1, 2 or 3).

For key events (CwKeyEvent):

keysym A constant describing the keyboard key that was pressed or released. These constants are found in the CwConstants pool dictionary, and are prefixed with `XK'.

character The Character describing the keyboard key that was pressed or released, or nil if it does not represent a valid character.

There are two common uses of event handlers. The first is for handling input in a drawing area widget. For example, in a graphical drawing application a drawing area widget would be used to display the drawing under construction. Event handlers would be registered to notify the application of pointer motion, mouse button, and key press events, allowing text strings to be edited and graphical objects to be positioned and changed using the mouse.

The second common use is for handling pop-up menus. An event handler is added for the *ButtonMenuMask* event. When the event handler is called, the application pops the menu up.

Mouse button 3 is used as the menu button. However, some platforms trigger the button menu event when the button is pressed, and others when the button is released. The ButtonMenuMask event hides this difference. It should be used, rather than the other button events, to support pop-up menus in a platform-independent manner.

Tip: On some platforms it is possible for a button release event to be delivered without a corresponding button press event. Applications should be prepared to ignore such spurious button release events by only processing a button release event that is received after a matching button press event.

In the example below, a drawing area is created, and an event handler is added to notify the application of mouse button presses, key presses, and pointer motion. Label widgets are used to display information about the events. The variable *labels* would be implemented as an instance variable for the class.

```
| shell rowColumn label drawing |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Event Handler Example’].
rowColumn := shell
   createRowColumn: ‘rowColumn’
   argBlock: nil.
rowColumn manageChild.
labels := Array new: 3.
1 to: 3 do: [:i |
   label := rowColumn
      createLabel: ‘label’
      argBlock: nil.
   label manageChild.
   labels at: i put: label].
(labels at: 1) labelString: ‘Position: ‘.
(labels at: 2) labelString: ‘Button pressed at position: ‘.
(labels at: 3) labelString: ‘Keysym of last pressed key: ‘.
drawing := rowColumn
   createDrawingArea: ‘drawing’
   argBlock: [:w |
      w
         borderWidth: 1;
         width: 300;
         height: 300].
drawing
   addEventHandler: ButtonPressMask | KeyPressMask | PointerMotionMask
   receiver: self
   selector: #eventHandler:clientData:event:
   clientData: nil.
drawing manageChild.
shell realizeWidget
```

When an event occurs, the following method is executed. Information about the event is determined from the event argument and is displayed in the label widgets.

```
eventHandler: widget clientData: clientData event: event
   “Handle an input event.”
   event type = MotionNotify
      ifTrue: [(labels at: 1) labelString: ‘Position: ‘,
            event point printString].
   event type = ButtonPress
      ifTrue: [(labels at: 2) labelString: ‘Button ‘,
            event button printString,
            ‘ pressed at position: ‘, event point printString].
   event type = KeyPress
      ifTrue: [(labels at: 3) labelString:
            ‘Keysym of last pressed key: ‘,
            event keysym printString].
```

Chapter 11 Common Widget Classes

The previous section provided an overview of how widgets are created and configured, and how they interact with an application. This section describes how to create and use specific widgets in the Common Widgets subsystem.

The following widgets are discussed:

- Shells (CwTopLevelShell, CwOverrideShell, CwTransientShell, CwDialogShell)
- Main windows and scrolled windows (CwMainWindow and CwScrolledWindow)
- Text editors and drawing areas (CwText, WbScrolledText and CwDrawingArea)
- Layout widgets (CwForm and CwRowColumn)
- Buttons and labels (CwLabel, CwDrawnButton, CwPushButton, CwToggleButton)
- Lists and combo boxes (CwList, WbScrolledList and CwComboBox)

Shell Widgets

Shell widgets provide the interface between an application and the platform’s window manager. The window manager is the part of the platform window system that manages the geometry, appearance, and stacking order of windows on the display. The window manager may add window decorations to a window, such as a frame, a title, resize handles, minimize and maximize buttons, and a close button. Window decorations are described in more detail in the section on “Top-Level Shell Widgets.” The window manager also keeps track of which window has input focus, that is, which window receives keyboard input. A shell can receive a focus callback when focus is either lost or gained.

A shell widget looks like a window on the screen. Shell widgets contain exactly one child. A CwTopLevelShell provides a normal window with standard appearance and decorations, and does not have a parent. CwTopLevelShell widgets are described in detail in the next section. CwOverrideShell, CwTransientShell, and CwDialogShell widgets must have a parent widget. These shells are described in this section.

CwOverrideShell widgets are used for pop-up windows that bypass window management and appear in front of all other windows. They do not have a window frame, and cannot be moved, resized or iconified by the user. CwOverrideShell widgets are created using the CwShell class method `createPopupShell:parent:argBlock:`. A CwOverrideShell is made visible sending it the popup message.

CwTransientShell widgets are pop-up windows that appear in front of all widgets in the tree of their parent widget. They have a window frame, and can be moved and resized (on some platforms), but cannot be iconified independently of their parent widget. When the parent widget's shell is iconified, the CwTransientShell is removed from the screen. A CwTransientShell is not usually instantiated directly; instead, an instance of its subclass, CwDialogShell, is created. A CwTransientShell is made visible sending it the popup message.

CwDialogShell widgets are pop-up windows used to implement modal or modeless dialog windows. The child of a CwDialogShell is typically an instance of a subclass of CwBulletinBoard. A CwDialogShell and its child are typically created automatically by using one of the dialog convenience methods. Unlike other types of shells, a CwDialogShell popped up by managing its child. The parent of a CwDialogShell can be any widget, and the dialog always appears over the window containing its parent widget. For further information on dialog shells and dialogs, see the section on "Composite Box Widgets" in the *IBM Smalltalk Programmer's Reference*.

Top-Level Shell Widgets

This section describes how to create and use top-level shell widgets (CwTopLevelShell). Some commonly used shell resources and callbacks are discussed.

The root of a widget tree must be a CwTopLevelShell. A top level shell widget has no parent. Top level shells are created using the `createApplicationShell:argBlock:` method, which is sent to the CwTopLevelShell class. A top level shell widget must have a child widget before it can be realized.

Tip: A common programming error is to attempt to realize a top level shell that has no child widget, or whose child widget computes an initial size with zero width or height. On a Motif platform this normally causes the application to exit with the message: "Error: Shell widget has zero width and/or height." VisualAge detects most common cases and prevents the application from exiting.

The following example creates a top level shell with a main window widget as its child. In this example, the main window's width and height are explicitly set. However a main window is not usually given explicit dimensions. It is usually left to calculate its dimensions based on the needs of its children.

```
| shell mainWindow |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Shell Example 1’].
mainWindow := shell
   createMainWindow: ‘main’
   argBlock: [:w | w width: 100; height: 100].
mainWindow manageChild.
shell realizeWidget.
```

The resources for CwTopLevelShell include *title*, *mwmDecorations*, *icon*, *iconPixmap* and *iconMask*. The *mwmDecorations* resource indicates which decorations are to be added to the shell. The following table lists the bit masks used to specify decorations. The *icon* (or *iconPixmap*) resources indicate the icon (or pixmap) to be used by the window manager for the application’s icon, and the *iconMask* resource indicates the pixmap to clip to if the icon is non-rectangular.

Decoration Literal	Definition
MWMDECORALL	If set, changes the meaning of the other flags to indicate that the specified decoration should be removed from the default set.
MWMDECORBORDER	Include a border
MWMDECORRESIZEH	Include resize frame handles
MWMDECORTITLE	Include title bar
MWMDECORMENU	Include window close/system menu
MWMDECORMINIMIZE	Include minimize window button
MWMDECORMAXIMIZE	Include maximize window button

TopLevel Shell Decoration Resource Flags

Note: The top level shell decorations settings indicate the preferred configuration for the window. The window manager can alter or ignore the settings if particular combinations are not supported by the platform.

In the example below, the shell’s mwmDecorations resource is explicitly altered in the create argBlock to specify that the minimize button should not be provided.

```
| shell main |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w |
      w
          title: ‘Shell Example 2’;
          mwmDecorations: MWMDECORALL | MWMDECORMINIMIZE].
```

```
main := shell
   createMainWindow: ‘main’
   argBlock: [:w |
      w
          width: 100;
          height: 100].
main manageChild.
shell realizeWidget.
```

Main Window Widgets

The main window widget (CwMainWindow) is used to organize the application’s menu bar and the widgets that define the application’s work region. The CwMainWindow class also includes all of the functionality provided by the CwScrolledWindow class and can provide scroll bars for scrolling the work region. If a main window is used, it must be the immediate child of a top level or dialog shell.

A main window widget is created by sending the `#createMainWindow:argBlock:` message to a shell widget.

A CwMainWindow must always be created as the child of a CwTopLevelShell or CwDialogShell. Creating it as the child of any other widget is an error.

Main Windows and Geometry Management

Like other composite widgets, a main window widget manages the geometry of its children. In order to manage its children correctly, it must know which widget is the menu bar, which widget is the work region, and which widgets are the scroll bars. The `#setAreas:horizontalScrollbar: verticalScrollbar:workRegion:` message explicitly tells the main window which of its child widgets are to be used for these purposes. In the example below, an empty menu bar and a drawing area widget are created as children of the main window. The `#setAreas:...` message is sent to the main window to explicitly set menuBar as the main window’s menu bar, and drawingArea as the main window’s work region. Because no scroll bars are being defined by the application, nil is passed in for the scroll bar arguments.

```
| shell main menuBar drawingArea |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: nil.
main := shell
   createMainWindow: ‘main’
   argBlock: nil.
main manageChild.
```

```
menuBar := main
   createSimpleMenuBar: ‘menu’
   argBlock: [:w | w buttons: #(‘’)].
menuBar manageChild.
drawingArea := main
   createDrawingArea: ‘draw’
   argBlock: [:w |
      w
         width: 300;
         height: 300].
drawingArea manageChild.
main
   setAreas: menuBar
   horizontalScrollbar: nil
   verticalScrollbar: nil
   workRegion: drawingArea.
shell realizeWidget.
```

Scrolled Window Widgets

The scrolled window widget (CwScrolledWindow) can scroll any other widget by positioning it behind a clipping area. No special processing by the application is required to implement scrolling. Any widget tree can be scrolled simply by making it the work region of a CwScrolledWindow or CwMainWindow widget.

A scrolled window widget is created by sending the `#createScrolledWindow:argBlock:` message to its parent. Next, the widget to be scrolled is created. It is made the work region for the scrolled window by using the `#setAreas:verticalScrollbar:workRegion:` message.

Scrolled window widgets support two scrolling policies, specified by the scrollingPolicy resource. These are XmAPPLICATIONDEFINED (the default) and XmAUTOMATIC. When the scrolling policy is XmAUTOMATIC, the scrolled window handles all aspects of scrolling, including creation of the scroll bars. The application can be notified of scroll bar movements by adding callbacks.

When the scrolling policy is XmAPPLICATIONDEFINED, the application must handle all aspects of scrolling, including creation of scroll bars. The scroll bars must be set using the `#setAreas:...` message.

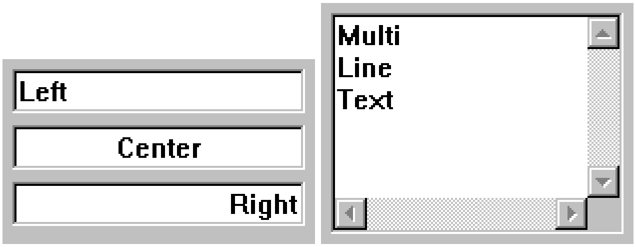
The scrollBarDisplayPolicy resource defines whether or not scrollbars are always showing (XmSTATIC) or displayed only if the work region exceeds the clip area (XmASNEEDED).

Note: The scrollingPolicy and scrollBarDisplayPolicy resources can only be set at creation time.

The following example creates a scrolled window containing several buttons in a vertical row-column. The scrolling policy is XmAUTOMATIC.

```
| shell scroll buttons |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Scrolled Buttons’].
scroll := shell
   createScrolledWindow: ‘scroll’
   argBlock: [:w | w scrollingPolicy: XmAUTOMATIC].
buttons := scroll
   createRowColumn: ‘buttons’
   argBlock: nil.
buttons manageChild.
(Collection withAllSubclasses collect: [:class | class name])
   asSortedCollection do: [:name |
      (buttons
         createPushButton: name
         argBlock: nil)
            manageChild].
scroll
   setAreas: nil
   verticalScrollbar: nil
   workRegion: buttons.
scroll manageChild.
shell realizeWidget.
```

Text Widgets



Left

Center

Right

Multi
Line
Text

Text widgets (CwText and WbScrolledText) provides text viewing and editing capabilities to the application. Text widgets can be created using the `#createText:argBlock:` and `#createScrolledText:argBlock:` convenience methods. The latter method makes the text scrollable, but otherwise provides basically the same functionality.

The entire contents of the text widget are set and retrieved using the `#setString:` and `#getString` methods.

When a scrolled text widget is created using `#createScrolledText:argBlock:`, a CwScrolledWindow widget is inserted between the CwText widget and the original parent. This is important to know when setting CwForm attachments, because in this case the attachments must be set on the text widget's parent (the scrolled window) rather than the text widget itself. WindowBuilder Pro provides a WbScrolledText widget that automatically sets up this structure and allows the programmer to differentiate between single and multi-line text edits (CwText and WbScrolledText respectively)

Two of the text widget's resources are *editMode* and *wordWrap*. The *editMode* resource specifies whether the widget supports single-line or multi-line editing of text. It can be set to XmSINGLELINEEDIT (the default for CwText) or XmMULTILINEEDIT (the default for WbScrolledText). The *wordWrap* resource specifies whether lines are to be broken at word breaks so that text does not go beyond the right edge of the window. The default setting for *wordWrap* is false.

Tip: Word wrap and horizontal scrolling are incompatible. In order for word wrap to work, the text widget must be configured without a horizontal scroll bar by setting the scrollHorizontal resource to false.

The example below creates a scrollable, multi-line text widget with word wrap on.

```
| shell text |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Text Widget Example’].
text := WbScrolledText
   createWidget: ‘text’
   parent: shell
   argBlock: [:w | w
      scrollHorizontal: false;
      wordWrap: true].
text setString: ‘Edit me!’.
text manageChild.
shell realizeWidget.
```

CwText and WbScrolledText widgets also have resources to control the initial number of rows and columns they contain, the position of the insertion point, the width of the tab character, and whether or not the widget is editable. CwText and WbScrolledText widgets can also set, get, cut, copy and paste a selection, scroll to a given line, and insert or replace text at a given position.

A text widget has **input focus** when it can accept keyboard input. The widget usually provides some visual indication that the it has focus, such as displaying the insertion position as a flashing I-beam or drawing a thicker border. Application programmers can add a *focusCallback* or a *losingFocusCallback* to a CwText or WbScrolledText if additional behavior is required when the widget either gains or loses focus.

Two other callbacks provided by text widgets are *modifyVerifyCallback*, called just before text is deleted from or inserted into the widget, and *valueChangedCallback*, called after text is deleted from or inserted into the widget. The example below uses a *modifyVerify* callback to allow only uppercase letters to be entered into a single-line CwText.

```
Object subclass: TextExample
 instanceVariableNames: ‘‘
 classVariableNames: ‘‘
 poolDictionaries: ‘CwConstants ‘
open
   | shell text |
   shell := CwTopLevelShell
      createApplicationShell: ‘shell’
      argBlock: [:w | w title: ‘Text Widget Example’].
   text := CwText
      createWidget: ‘test’
      parent: shell
      argBlock: [:w | w columns: 18].

   text
      addCallback: XmNmodifyVerifyCallback
      receiver: self
      selector: #modifyVerify:clientData:callData:
      clientData: nil.
   text manageChild.
   shell realizeWidget.

modifyVerify: widget clientData: clientData callData: callData

   “Update the stored version of the string in the callData, so
   that the text widget inserts capital letters instead of the
   real text typed or pasted by the user.”
      callData text: callData text asUppercase
```

Drawing Area Widgets

The drawing area (CwDrawingArea) widget provides an application with an area in which application-defined graphics can be drawn using Common Graphics operations such as `#fillRectangle`:, `#drawArc`:, and `#drawString`:. Consult the Common Graphics chapter of the *IBM Smalltalk Programmer’s Reference* for an explanation of drawing and other graphics operations.

Drawing is actually done on the CgWindow associated with the CwDrawingArea. Every CwWidget has a corresponding CgWindow, obtained with aCwWidget window, that can be used for drawing. Although any widget can be drawn on in this manner, CwDrawingArea widgets are typically used since they provide additional drawing-related functionality. CwDrawingArea widgets can be created using the `#createDrawingArea:argBlock:` convenience method.

A CwDrawingArea can be told to notify the application with an **expose** callback whenever a part of the drawing area needs to be redrawn. The expose callback contains an expose event with a rectangle describing the damaged area of the widget's CgWindow.

The example below is a simple application that draws a mandala. (A mandala is a drawing of lines connecting each of a given number of points on the circumference of a circle to every other such point.) Four callbacks that are often used in conjunction with drawing areas are illustrated: exposeCallback (described above), resizeCallback, inputCallback and destroyCallback.

The **resize** callback is called when the drawing area changes size, usually due to a change in the size of a parent widget. If an expose callback is triggered as a result of a resize, the resize callback is always sent before the expose callback. It is possible for the resize callback to be executed before the window has been realized. The resize callback handler should handle the case where the window message returns nil.

The **input** callback is called when a mouse button is pressed or released inside the widget or a key on the keyboard has been pressed or released. The **destroy** callback, executed when the widget is about to be destroyed, is a good place to free any graphics resources that have been allocated for drawing.

```
Object subclass: DrawingAreaExample
 instanceVariableNames: ‘gc radius segments ‘
 classVariableNames: ‘‘
 poolDictionaries: ‘CwConstants CgConstants ‘

example1
   “Open the drawing area example.”
   | diameter shell draw |
   radius := 150.
   diameter := radius * 2.
   shell := CwTopLevelShell
      createApplicationShell: ‘shell’
      argBlock: [:w | w title: ‘Drawing Area Example’].
   draw := CwDrawingArea
      createWidget: ‘draw’
      parent: shell
      argBlock: [:w | w
         width: diameter;
         height: diameter].
   draw manageChild.
   draw
      addCallback: XmNexposeCallback
         receiver: self
         selector: #expose:clientData:callData:
         clientData: nil;
      addCallback: XmNresizeCallback
         receiver: self
         selector: #resize:clientData:callData:
         clientData: nil;
      addCallback: XmNinputCallback
         receiver: self
         selector: #input:clientData:callData:
         clientData: nil;
      addCallback: XmNdestroyCallback
         receiver: self
         selector: #destroy:clientData:callData:
         clientData: nil.
   shell realizeWidget.
   gc := draw window
      createGC: None
      values: nil.
   gc setForeground: draw window blackPixel.

recalculateSegments: widget
   “Recalculate the coordinates of the mandala’s line segments.”
   | n points x y |
   n := 20.
   points := OrderedCollection new.
   0 to: Float pi * 2 by: Float pi * 2 / n do: [:angle |
      x := (angle cos * radius) rounded + (widget width // 2).
      y := (angle sin * radius) rounded + (widget height // 2).
      points add: x@y].
   segments := OrderedCollection new.
   1 to: points size - 1 do: [:i |
      i + 1 to: points size do: [:j |
      segments add:
       (CgSegment
         point1: (points at: i)
         point2: (points at: j))]].

expose: widget clientData: clientData callData: callData
   “Redraw the contents of the drawing area.”
   callData event count = 0
      ifTrue: [
         segments isNil
            ifTrue: [self recalculateSegments: widget].
         widget window
```

```
                drawSegments: gc
                segments: segments].

resize: widget clientData: clientData callData: callData
   “The drawing area has been resized.”
   widget window notNil
      ifTrue: [
         radius := (widget width min: widget height) // 2.
         segments := nil].

input: widget clientData: clientData callData: callData
   “The drawing area has received an input callback (button or
    key event).
    Explicitly destroy the widget if one of three things has
   happened:
      - the user typed ‘Q’ or ‘q’.
      - the user typed ‘control-DownArrow’.
      - the user did a ‘shift-click’ (shift key pressed, click
       left mouse button).”
   | event quit |
   quit := false.
   event := callData event.
   “$Q, $q, or control-End typed”
   event type = KeyPress
      ifTrue: [
         quit := (‘Qq’ includes: event character)
            or: [(event state & ControlMask) = ControlMask
                  and: [event keysym = XKdownarrow]]].
   “shift-click”
   (event type = ButtonPress and: [event button = 1])
      ifTrue: [
         quit := (event state & ShiftMask) = ShiftMask].
   quit ifTrue: [widget destroyWidget].

destroy: widget clientData: clientData callData: callData
 “The drawing area has been destroyed.
  Free any allocated graphics resources.”
   gc freeGC.
```

Adding an Event Handler to a Drawing Area

In the following example, a button press event handler is used to detect double-clicks in a drawing area. The open method creates the widgets, and the `#buttonPress:clientData:event:` method handles button press events.

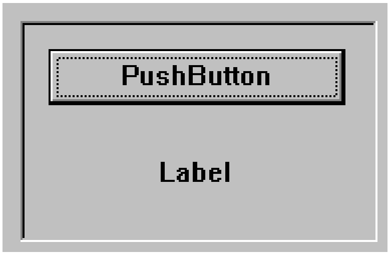
```
Object subclass: #DoubleClick
 instanceVariableNames: ‘clickStartTime ‘
 classVariableNames: ‘‘
 poolDictionaries: ‘CgConstants CwConstants ‘

open
   “Create a drawing area inside a shell.”
   | shell drawingArea |
   clickStartTime := 0.
   shell := CwTopLevelShell
      createApplicationShell: ‘shell’
      argBlock: [:w | w title: ‘Double-click test’].
   (drawingArea := CwDrawingArea
      createWidget: ‘draw’
      parent: shell
      argBlock: [:w | w
         width: 100;
         height: 100])
      manageChild.
   drawingArea
      addEventHandler: ButtonPressMask
      receiver: self
      selector: #buttonPress:clientData:event:
      clientData: nil.
   shell realizeWidget.

buttonPress: widget clientData: clientData event: event
   “Detect double click by checking whether the time between
    successive presses of the left mouse button is less
    than the system-defined double-click time.”
   event button = 1
      ifTrue: [
         event time - clickStartTime <
         widget display doubleClickInterval
            ifTrue: [
               clickStartTime := 0.
               Transcript cr; show: ‘DOUBLE CLICK’ ]
            ifFalse: [
               clickStartTime := event time ]].
```

Tip: Adding a mouse down event handler to a widget that processes mouse events internally, such as a CwPushButton, may result in unpredictable behavior. To detect double-clicks in a WbScrolledList, use the defaultAction callback.

Layout Widgets

The form (CwForm) and row-column (CwRowColumn) widgets are composite widgets that allow the application to specify how child widgets of the composite should be laid out relative to each other and relative to the composite.

Form Widgets



Form widgets can be created using the `#createForm:argBlock:` convenience method. Form widget children are positioned by attaching their sides to other objects. Attachments are specified by setting each child's leftAttachment, rightAttachment, topAttachment and bottomAttachment resources. A side can be attached either to a given position, to another widget, or to the edge of the form. The attachment types are listed below. The first four types are the most commonly used. All are described in terms of the leftAttachment, but the same attachment types apply to the other sides, with corresponding behavior.

XmATTACHNONE

Default. Do not attach this side.

XmATTACHFORM

Attach the left side of the child to the left side of the form.

XmATTACHWIDGET

Attach the left side of the child to the right side of the widget specified in the leftWidget resource.

XmATTACHPOSITION

Attach the left side of the child to a relative position in the form. This position is specified by the leftPosition resource, and is a fractional value of the width of the form, with the default range being from 0 to 100. The position is relative to the left side of the form for left and right attachments, and to the top of the form for top and bottom attachments. A position of 0 places the left side of the child at the left side of the form. A position of 100 places the left side of the child at the right side of the form.

XmATTACHOPPOSITEFORM

Attach the left side of the child to the right side of the form.

XmATTACHOPPOSITEWIDGET

Attach the left side of the child to the left side of the widget specified in the leftWidget resource.

XmATTACHSELF

Attach the left side of the child to its initial position in the form.

Note: It is an error for attachments to be recursively defined. For example, if a widget A is attached to a widget B, then widget B cannot be attached to widget A. More generally, there must not be a cycle in the widget attachments.

If the attachment is XmATTACHFORM or XmATTACHWIDGET, an offset can also be specified that adds space between the side of the widget and the object to which it is attached. Offsets are specified by the leftOffset, rightOffset, topOffset and bottomOffset resources. Offsets are specified in units of pixels.

Note: The results are undefined if an offset setting is used with an attachment type of XmATTACHPOSITION.

If attachments have been set on all sides of a widget, the size of the widget is completely determined by the form and the other child widgets. However, if a side is left unattached, the widget will use its preferred size in the corresponding dimension. This is useful for allowing widgets to size themselves automatically based on their font size, contents, and other attributes.

Some convenience methods, such as those used to create a scrolled list or a scrolled text, actually create a widget sub-tree, but instead of returning the root of the sub-tree, the child is returned. In these cases, the form attachments must be set on the returned widget's parent, rather than on the widget itself.

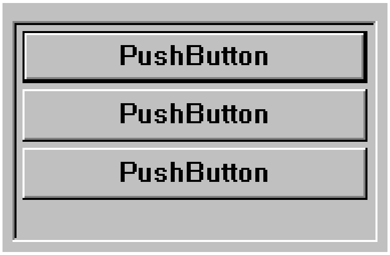
The example below illustrates a form containing a drawing area and a text widget. The right side of the drawing area is attached to a position two-thirds (67 per cent) of the way from left to right. The left side of the text widget is attached to the right side of the drawing area. The remaining sides of the text and drawing area widgets are attached to the form. The widgets are offset from each other by two pixels. (Offsets in the diagram have been exaggerated to show the attachments.)

The following code example creates the widget tree illustrated above.

```
| shell form drawing text |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Form Example’].
form := shell
```

```
   createForm: ‘form’
   argBlock: nil.
form manageChild.
drawing := form
   createDrawingArea: ‘drawing’
   argBlock: [:w |
      w
         borderWidth: 1;
         width: 200;
         height: 200;
         leftAttachment: XmATTACHFORM;
         leftOffset: 2;
         rightAttachment: XmATTACHPOSITION;
         rightPosition: 67;
         topAttachment: XmATTACHFORM;
         topOffset: 2;
         bottomAttachment: XmATTACHFORM;
         bottomOffset: 2].
drawing manageChild.
text := form
   createText: ‘text’
   argBlock: [:w |
      w
         leftAttachment: XmATTACHWIDGET;
         leftWidget: drawing;
         leftOffset: 2;
         rightAttachment: XmATTACHFORM;
         rightOffset: 2;
         topAttachment: XmATTACHFORM;
         topOffset: 2;
         bottomAttachment: XmATTACHFORM;
         bottomOffset: 2].
text manageChild.
shell realizeWidget.
```

Row-Column Widgets



The row-column widget (CwRowColumn) positions its children in rows or columns. CwRowColumn widgets are frequently used to lay out groups of buttons, including pop-up and pulldown menus. They can also be used to lay out widgets in a table. Row-column widgets can be created using the `#createRowColumn:argBlock:` convenience method.

Some commonly used row-column resources are the *orientation*, *marginWidth*, *marginHeight* and *spacing* resources. The *orientation* resource specifies that the layout is either row major or column major. In a column major layout, specified by XmVERTICAL, the children are laid out in columns top to bottom. In a row major layout, specified by XmHORIZONTAL, the children are laid out in rows. The default orientation is XmVERTICAL. The *marginWidth* and *marginHeight* resources specify the size of the margin between the child widgets and the edges of the row-column. The *spacing* resource specifies the spacing between child widgets.

In the illustration below, the buttons on the left are organized in a row-column widget. The row-column and the drawing area are contained in a form, similar to the previous example.

The following code creates the example shown above.

```
| shell form rowColumn drawing |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘RowColumn Example’].
form := shell
   createForm: ‘form’
   argBlock: nil.
form manageChild.
rowColumn := form
   createRowColumn: ‘rooms’
   argBlock: [:w |
      w
         orientation: XmVERTICAL;
         marginWidth: 10;
         marginHeight: 10;
         spacing: 20;
         leftAttachment: XmATTACHFORM;
         topAttachment: XmATTACHFORM;
         bottomAttachment: XmATTACHFORM].
rowColumn manageChild.
#(‘Kitchen’ ‘Dining Room’ ‘Living Room’ ‘Washroom’ ‘Bedroom’
‘Workshop’)
   do: [:room |
      (rowColumn
         createPushButton: room
         argBlock: nil)
            manageChild].
drawing := form
   createDrawingArea: ‘drawing’
   argBlock: [:w |
      w
         borderWidth: 1;
         width: 300;
         leftAttachment: XmATTACHWIDGET;
         leftWidget: rowColumn;
         leftOffset: 2;
         rightAttachment: XmATTACHFORM;
         rightOffset: 2;
         topAttachment: XmATTACHFORM;
         topOffset: 2;
         bottomAttachment: XmATTACHFORM;
         bottomOffset: 2].
drawing manageChild.
shell realizeWidget.
```

Button and Label Widgets

The Common Widgets subsystem allows applications to create static text labels (CwLabel) and several types of buttons:

- Push buttons (CwPushButton)
- On/off toggle buttons (CwToggleButton)
- Application-drawn buttons (CwDrawnButton)

Buttons and labels can display either strings, pixmaps or icons as their contents, depending on the value of the *labelType* resource.

The following resources define the visual appearance of labels and buttons: *x*, *y*, *height*, *width*, *marginTop*, *marginBottom*, *marginHeight*, *marginWidth*, *marginLeft* and *marginRight*.

The *marginTop*, *marginBottom*, *marginRight* and *marginLeft* resources are typically controlled by subclasses of CwLabel or by the label's parent. For example, a CwToggleButton could increase *marginRight* to make space for the toggle indicator. The *marginHeight* and *marginWidth* resources are usually left alone by subclasses, and can be manipulated by the application if desired.

Tip: The margin resource settings indicate the preferred appearance of the widget. They may be ignored if they are not supported by the platform or conflict with the platform's look and feel.

By default, the name given in a label or button creation message is used as the widget's *labelString*. The contents of a label or button widget are changed using the `#labelString:` resource method.

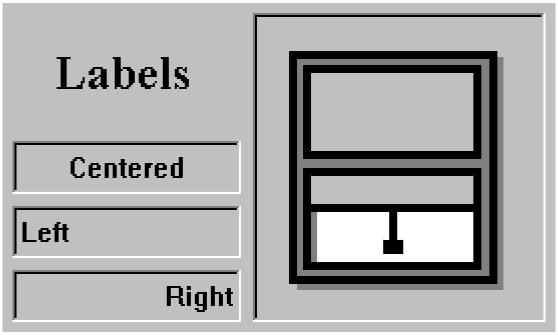
CwLabel provides *accelerator* and *acceleratorText* resources for adding an accelerator key to a toggle button or push button that is in a popup or pulldown menu. An accelerator key will activate a button at any time, provided the parent menu is managed. The *accelerator* resource is set to an instance of CwAccelerator created using the `#mask:keysym:` class method, which takes the following arguments:

mask The modifier key mask. Consists of a logical-or of zero of more of the following: Mod1Mask, ControlMask and/or ShiftMask.

keysym The unmodified key, which must be a lowercase letter or special key, represented by a CwConstants 'XK' keysym value.

The *acceleratorText* resource describes the string that is displayed beside the button in the menu.

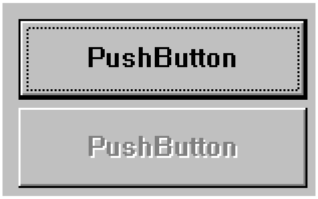
Static Label Widgets



Static label widgets (CwLabel) can be created using the `#createLabel:argBlock:` convenience method. Static labels do not provide any special callbacks. The following code creates an simple example.

```
| shell label |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Label Example’].
label := shell
   createLabel: ‘label’
   argBlock: nil.
label labelString: ‘This is a label.’.
label manageChild.
shell realizeWidget.
```

Push Button Widgets



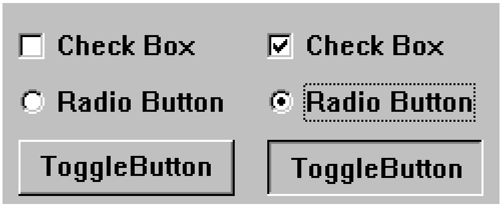
Push button widgets (CwPushButton) can be created using the `#createPushButton:argBlock:` convenience method.

Push buttons call their *activate* callback when they are pressed and released.

In the example below, three buttons are created in a row-column. An activate callback has been added to each button. The same callback message is used in all three cases. The client data of the callback is used to identify which button was pressed.

```
| shell rowColumn row b1 b2 b3 |
shell := CwTopLevelShell
   createApplicationShell: ‘Test’
   argBlock: nil.
rowColumn := shell
   createRowColumn: ‘buttons’
   argBlock: nil.
rowColumn manageChild.
b1 := rowColumn
   createPushButton: ‘Top’
   argBlock: nil.
b1
   addCallback: XmNactivateCallback
   receiver: self
   selector: #button:clientData:callData:
   clientData: ‘top’.
b1 manageChild.
b2 := rowColumn
   createPushButton: ‘Middle’
   argBlock: nil.
b2
   addCallback: XmNactivateCallback
   receiver: self
   selector: #button:clientData:callData:
   clientData: ‘middle’.
b2 manageChild.
b3 := rowColumn
 createPushButton: ‘Bottom’
 argBlock: nil.
b3
   addCallback: XmNactivateCallback
   receiver: self
   selector: #button:clientData:callData:
   clientData: ‘bottom’.
b3 manageChild.
shell realizeWidget.
The activate callback used in the code is shown below.
button: widget clientData: clientData callData: callData
   “A button has been pressed.”
   Transcript cr; show: ‘The ‘, clientData,
      ‘ button has been pressed.’
```

Toggle Button Widgets

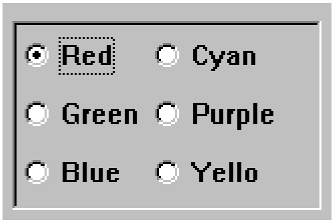


Toggle button widgets (CwToggleButton) can be created using the `#createToggleButton:argBlock:` convenience method.

Toggle buttons have two states: *on* and *off*. The state of a toggle button can be queried and changed using the `#getState` and `#setState:notify:` messages, respectively. Toggle buttons call their valueChanged callback when their state is changed.

Toggle buttons are typically used to create radio button and check box groups using row column convenience methods described in the next sections. The toggle button *indicatorType* resource controls whether the toggle button has a radio button or a check box appearance. When the resource value is set to XmONEOFMANY, the button has a radio button appearance. When the value is set to XmNOFMANY, the button has a check box appearance.

Radio Button Groups



A row-column widget containing several toggle button widgets (CwToggleButton) can be configured to have radio button behavior. When a button is selected in this mode, any other selected buttons in the group are automatically deselected, leaving only one button selected at any time. The *radioBehavior* resource of the CwRowColumn widget controls this behavior.

A CwRowColumn with *radioBehaviour* set to true is created using the convenience method `#createRadioBox:argBlock:`. WindowBuilder Pro provides a WbRadioBox widget that automatically sets up this behavior.

Tip: As a side effect of `#createRadioBox:argBlock:`, the CwRowColumn’s isHomogeneous resource is set to true. Children of a homogeneous row-column widget must all be of the same type. In this case, they must all be CwToggleButton widgets.

A toggle button can be selected or deselected using the `#setState:notify:` method. Its state can be queried using the `#getState` method. The valueChanged callback of a toggle button is executed whenever the state of the button changes.

Tip: The valueChanged callback is executed when a button is deselected as well as when it is selected. The state of the widget should be checked in the callback using the `#getState` method, or by checking the set field of the callback data.

In the example below, a radio box row-column is created. Three toggle buttons are added. The same valueChanged callback is added to each toggle button, with the client data used to identify the selected button. The resulting radio button group is shown in the left margin. For simplicity, the shell is not shown.

```
| shell rowColumn button buttonNames initialValues languageNames|
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Radio Box Example’].
rowColumn := shell
   createRadioBox: ‘radio’
   argBlock: nil.
rowColumn manageChild.
buttonNames := #(‘Hello’ ‘Bonjour’ ‘Ola’).
initialValues := (Array with: true with: false with: false).
languageNames := #(‘English’ ‘Franch’ ‘Spanish’).
1 to: buttonNames size
   do: [:i |
      button := rowColumn
         createToggleButton: (buttonNames at: i)
         argBlock: [:w | w set: (initialValues at: i)].
      button
         addCallback: XmNvalueChangedCallback
         receiver: self
         selector: #language:clientData:callData:
         clientData: (languageNames at: i).
      button manageChild].
shell realizeWidget.
```

The *valueChanged* callback used by the code is shown below. The selected language is indicated by the *clientData* argument. A message is written to the transcript whenever a new language is chosen.

```
language: widget clientData: clientData callData: callData
   “A toggle button has changed state.”
   callData set
      ifTrue: [Transcript cr; show:
         ‘The selected language is now ‘, clientData, ‘.’].
```

Check Boxes

Check boxes consist of several toggle buttons that present a set of options to the user. The user can choose none, all, or any combination of the buttons. A CwRowColumn widget can be used to contain the buttons. When the row-column’s *radioBehavior* resource is false, its default, more than one toggle button can be selected at a time.

```
The code that follows creates a toggle button group.
| shell rowColumn button buttonNames initialValues |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Check Box Example’].
rowColumn := shell
   createRowColumn: ‘group’
   argBlock: nil.
rowColumn manageChild.
buttonNames := #(‘Item 1’ ‘Item 2’ ‘Item 3’).
initialValues := (Array with: false with: true with: true).
1 to: buttonNames size : [:i |
   button := rowColumn
      createToggleButton: (buttonNames at: i)
      argBlock: [:w | w set: (initialValues at: i)].
   button
      addCallback: XmNvalueChangedCallback
      receiver: self
      selector: #valueChanged:clientData:callData:
      clientData: nil.
   button manageChild].
shell realizeWidget.
```

The *valueChanged* callback used by the code is shown below. A message is written to the transcript whenever a button is selected or deselected.

```
valueChanged: widget clientData: clientData callData: callData
   “A toggle button has changed state.”
   Transcript cr; show: widget labelString, ‘ has been ‘.
   callData set
      ifTrue: [Transcript show: ‘selected.’]
      ifFalse: [Transcript show: ‘deselected.’].
```

Icon and Pixmap Label and Button Widgets

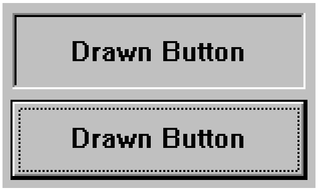
The contents of CwLabel, CwPushButton, and CwToggleButton widgets can be a string, an icon or a pixmap. When the *labelType* resource is XmSTRING, the labelString resource specifies the string to display. The default type is string. When labelType is XmICON, labelIcon specifies the icon to use, and when *labelType* is XmPIXMAP, labelPixmap specifies the pixmap to display. Consult "Using Pixmaps" in the *IBM Smalltalk Programmer's Reference* for more information on using pixmaps and icons.

The code below creates a widget tree containing the pixmap button shown at left. Icon buttons and labels are created in a similar manner. Note that *pixmap* is an instance variable.

```
| shell button questionMark |
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Pixmap Button Example’].
button := shell
   createPushButton: ‘button’
   argBlock: nil.
button
   addCallback: XmNdestroyCallback
   receiver: self
   selector: #destroy:clientData:callData:
   clientData: nil.
questionMark := #(0 0 0 0 255 0 192 255 3 224 255 7
   240 255 15 248 255 31 248 255 31 252 255 63
   252 227 63 252 193 63 252 193 63 252 193 63
   248 224 63 248 240 31 0 248 31 0 252 15
   0 252 7 0 254 3 0 254 3 0 254 1
   0 254 1 0 252 0 0 252 0 0 0 0
   0 252 0 0 254 1 0 254 1 0 254 1
   0 254 1 0 252 0 0 0 0 0 0).
   “Realize the shell without mapping it so we have access to the
   button’s window & palette without making the it appear.”
   shell
      mappedWhenManaged: false;
      realizeWidget.
pixmap := button screen rootWindow
   createPixmapFromBitmapData: questionMark
   width: 24
   height: 32
   fg: button window blackPixel
   bg: (button window getPalette
      nearestPixelValue: button backgroundColor)
   depth: button depth.
```

```
button
   setValuesBlock: [:w |
      w
         labelType: XmPIXMAP;
         labelPixmap: pixmap].
button manageChild.
shell mapWidget.
destroy: widget clientData: clientData callData: callData
   pixmap freePixmap.
```

Application-Drawn Buttons



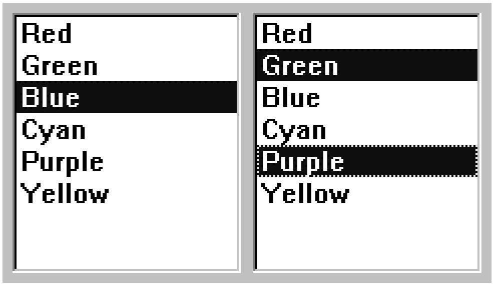
Application-drawn button widgets (CwDrawnButton) enable the application to draw arbitrary graphics on a button. Drawn buttons behave like push buttons except that they can be drawn on like drawing area widgets. See the example below.

As with the push button widget, the application can add an activate callback to be executed when the button is pressed. As with the drawing area widget, *expose* and *resize* callbacks can be added to notify the application when the button requires redrawing and when it has changed size. Consult “Drawing Operations” in the *IBM Smalltalk Programmer’s Reference* for more information on drawing graphics.

In the code below, the drawn button shown at left is created and drawn.

```
Object subclass: #DrawnButtonExample
 instanceVariableNames: ‘gc ‘
 classVariableNames: ‘‘
 poolDictionaries: ‘CwConstants CgConstants ‘

open
   | shell button |
   shell := CwTopLevelShell
      createApplicationShell: ‘shell’
      argBlock: [:w | w title: ‘Drawn Button Example’].
   button := shell
      createDrawnButton: ‘button’
      argBlock: nil.
   button
      addCallback: XmNactivateCallback
      receiver: self
      selector: #button:clientData:callData:
      clientData: nil;
      addCallback: XmNexposeCallback
      receiver: self
      selector: #expose:clientData:callData:
      clientData: nil;
      addCallback: XmNdestroyCallback
      receiver: self
      selector: #destroy:clientData:callData:
      clientData: nil.
   button manageChild.
   shell realizeWidget.
   gc := button window
      createGC: None
      values: nil.

activate: widget clientData: clientData callData: callData
   “The drawn button has been pressed.”
   Transcript cr; show: ‘The pixmap button has been pressed.’.

expose: widget clientData: clientData callData: callData
   “The drawn button has been exposed. Redraw the button.”
   | x |
   callData event count = 0
      ifTrue: [
         0 to: 10 do: [:i |
            x := widget width * i // 10.
            widget window
               drawLine: gc
               x1: x
               y1: 0
               x2: widget width - x
               y2: widget height - 1]].

destroy: widget clientData: clientData callData: callData
   gc freeGC.
```

List Widgets



List widgets (CwList and WbScrolledList) present a list of items and allow the user to select one or more items from the list. List widgets can be created using the `#createList:argBlock:` and `#createScrolledList:argBlock:` convenience methods. The latter method makes the list scrollable, but otherwise provides basically the same functionality. WindowBuilder Pro provides a widget, WbScrolledList, that automatically sets up the scrollable behavior.

The items in the list and the selected items are specified by the items and selectedItems resources, respectively. The selectionPolicy resource specifies the policy for selecting items. It has four possible settings:

XmBROWSESELECT Allows only single selection. Behavior may vary from platform to platform, but normally the selection moves when the mouse is dragged. This is the default selection policy.

XmSINGLESELECT Allows only single selection. Behavior may vary from platform to platform, but normally the selection remains the same when the mouse is dragged.

XmMULTIPLESELECT Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Behavior may vary from platform to platform, but normally clicking on an item without a modifier key held down deselects all previously selected items.

Tip: On some platforms, browse select and single select work the same way.

List widgets provide several methods for adding, deleting and replacing items and selected items in the list.

The selectionPolicy resource determines which callback is used to notify the application of changes in the selection. List widgets support the following callbacks:

browseSelectionCallback Executed when an item is selected in browse selection mode

singleSelectionCallback Executed when an item is selected in single selection mode

multipleSelectionCallback Executed when an item or group of items is selected in multiple selection mode

extendedSelectionCallback Executed when an item or group of items is selected in extended selection mode

defaultActionCallback Executed when an item is double clicked (all modes)

The call data of the selection callback specifies the item or items that were selected, and the position(s) of the selected item(s) in the list. Item positions in the list are numbered starting from one.

Single Selection Lists

In the example below, the list widget shown at left is created with its selection policy set to XSINGLESELECT. A singleSelection callback is added, to correspond with the selection policy.

```
| items shell list |
items := #(‘item1’ ‘item2’ ‘item3’ ‘item4’ ‘item5’ ).
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: nil.
list := WbScrolledList
   createWidget: ‘list’
   parent: shell
   argBlock: [:w | w
      selectionPolicy: XmSINGLESELECT;
      items: items].
list
   addCallback: XmNsingleSelectionCallback
   receiver: self
   selector: #singleSelect:clientData:callData:
```

```
   clientData: nil.
list manageChild.
shell realizeWidget.
```

The call data of the singleSelection callback specifies the item that was selected. The callback method below prints the entire callback data on the transcript. All components of the call data can be retrieved using the corresponding accessor method.

```
singleSelect: widget clientData: clientData callData: callData
   “Print the call data.”
   Transcript cr; show: ‘Single selection call data: ‘,
      callData printString
```

If Item 2 was selected, as in the illustration, the transcript output would be:

```
   Single selection call data: CwListCallbackData(
    reason -> 23
    item -> ‘Item 2’
    itemPosition -> 2
    selectedItems -> nil
    selectedItemCount -> nil
    selectedItemPositions -> nil
```

Multiple Selection Lists

In the example below, the list widget shown at left is created with its selection policy set to XmMULTIPLESELECT. A multipleSelection callback is added, to correspond with the selection policy.

```
| items shell list |
items := #(‘item1’ ‘item2’ ‘item3’ ‘item4’ ‘item5’ ).
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: nil.
list := WbScrolledList
   createWidget: ‘list’
   parent: shell
   argBlock: [:w | w
      selectionPolicy: XmMULTIPLESELECT;
      items: items].
list
   addCallback: XmNmultipleSelectionCallback
   receiver: self
   selector: #multipleSelect:clientData:callData:
   clientData: nil.
list manageChild.
shell realizeWidget.
```

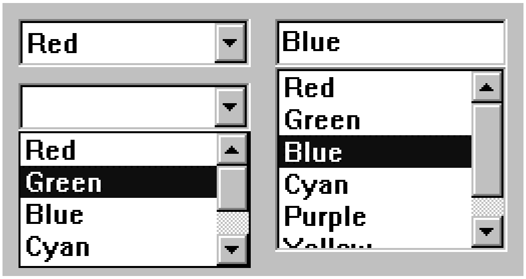
The call data of the multipleSelection callback specifies the items that were selected. The callback method below prints the entire callback data on the transcript. All components of the call data can be retrieved using the corresponding accessor method.

```
multipleSelect: widget clientData: clientData callData: callData
   “Print the call data.”
   Transcript cr; show: ‘Multiple selection call data: ‘,
      callData printString
```

If Item 2 and Item 3 were selected in order, as in the illustration, the transcript output would be:

```
   Multiple selection call data: CwListCallbackData(
    reason -> 24
    item -> ‘Item 3’
    itemPosition -> 3
    selectedItems -> OrderedCollection (‘Item 2’ ‘Item 3’)
    selectedItemCount -> 2
    selectedItemPositions -> OrderedCollection(2 3))
```

Combo Box Widgets



Like list widgets, combo box widgets (CwComboBox) enable the user to select from a list of available items. A combo box also displays the last selected item in a text box above the list. Combo box widgets can only have one item selected at a time. Combo box widgets can be created using the `#createComboBox:argBlock:` convenience method.

There are two styles of combo boxes, specified by the comboBoxType resource:

XmDROPDOWN The list is displayed only when dropped down by pressing a button beside the text box. When a selection is made, the list disappears (default).

XmSIMPLE The list is always displayed

As with the list widget, the items in the combo box are specified by the items resource. The application can add a *singleSelection* callback to be executed whenever the selection changes. Several methods are provided for adding, deleting, and replacing items in the list.

The contents of the text part of the combo box can be set and retrieved using the `#setString:` and `#getString` methods.

The following example creates the drop down combo box shown at left. Its items are set, the contents of the text box are initialized to the first item, and a singleSelection callback is added.

```
| items shell combo |
items := #(‘Item 1’ ‘Item 2’ ‘Item 3’ ‘Item 4’).
shell := CwTopLevelShell
   createApplicationShell: ‘shell’
   argBlock: [:w | w title: ‘Combo Box Example’].
combo := CwComboBox
   createWidget: ‘combo’
   parent: shell
   argBlock: [:w | w
      comboBoxType: XmDROPDOWN;
      items: items].
combo setString: items first.
combo
   addCallback: XmNsingleSelectionCallback
   receiver: self
   selector: #singleSelect:clientData:callData:
   clientData: nil.
combo manageChild.
shell realizeWidget.
```

Chapter 12 Widget Encyclopedia

This chapter provides descriptions of each method and callback that is understood by the widgets supported by WindowBuilder Pro. The first section describes protocols that are understood by all widgets. Each following section describes the protocols understood by a particular widget class. For resource get and set protocols, only the set version of the protocol has been described. For each set protocol (e.g., `#height`:, `#x`:, `#borderWidth`:, etc.), there is a corresponding get protocol (e.g., `#height`, `#x`, `#borderWidth`, etc.).

All Widgets

Protocol

addCallback: *callbackName* **receiver:** *receiver* **selector:** *selector*
clientData: *clientData*
Add a callback to one of the receiver's callback lists. Generally speaking, a widget expecting to interact with an application will declare one or more callback lists as resources; the application adds callbacks to these callback lists, which will be invoked whenever the predefined callback conditions are met. Callback lists are resources, so that the application can set or change the function that will be invoked.

Callbacks are not necessarily invoked in response to any event; a widget can call the specified routines at any arbitrary point in its code, whenever it wants to provide a 'hook' for application interaction. For example, all widgets provide a destroyCallback resource to allow applications to interpose a routine to be executed when the widget is destroyed.

This message adds a new callback to the end of the callback list. A callback will be invoked as many times as it occurs in the callback list.

callbackName	The resource name of the callback list to which the callback is to be appended.
receiver	The object to send the callback message to.

selector	The 3-parameter message selector to send (WindowBuilder Pro also supports unary message selectors).
clientData	An object to be passed to the receiver of the callback message as the clientData parameter when the callback is invoked, or nil.

addEventHandler: *eventMask* **receiver:** *receiver* **selector:** *selector*
clientData: *clientData*
Register an event handler. This message registers with the dispatch mechanism. The handler thus registered will be called when an event matching the eventMask occurs in the receiver. This message can be sent at any time during the widget's lifetime.

A handler may be registered with the same clientData to handle multiple events. Further, more than one event handler can be registered for a given event. If multiple handlers are registered, the handlers will all be called, but in an indeterminate order.

NOTE:

1) Event handlers are not supported for menu bars, menus or any widget that is part of a menu.

2) Event handlers are not supported for 'private' pseudo-widgets that make up the implementation of widgets such as scrolled text, scrolled list, combo box, scale, and scrolled window. Depending on the particular platform, such private widgets may or may not actually exist.

3) Exposure events are not supported through the event handler mechanism. Widgets that support exposure notification (e.g. CwDrawingArea and CwDrawnButton) provide callbacks for this purpose.

4) Since the proper event to trigger a menu popup varies among different platforms, the ButtonMenuMask is defined to select for the correct event for the platform, and should always be used for event handlers which pop up menus.

eventMask	An integer valued event mask specifying the events interest. The eventMask parameter is constructed as a bitwise OR of the individual event masks.
receiver	The object to send the event handler message to.
selector	The 3-parameter message selector to send (WindowBuilder Pro also supports unary message selectors).
clientData	An object to be passed to the receiver of the event handler as the clientData parameter when the event handler is invoked, or nil.

allChildren
Answer a collection of all of the receiver's children.

allMajorChildren
Answer a collection of all of the receiver's major children. This collection excludes widgets such as the scrollbar children of a scrolled list.

allParents
Answer a collection of the receiver's parents until the top level shell is reached.

ancestorSensitive
Specifies whether the immediate parent of the widget will react to input events.

backgroundColor: *aCgRGBColor*
Specifies the background drawing color. **NOTE:** The particular aspects of the widget's appearance which are affected by changing this resource are dependent on platform-specific styles and capabilities and vary from platform to platform.

basicWidget
Answer the receiver's basic widget. The basic widget is any instance of a subclass of CwBasicWidget.

Extended widgets are implemented using basic widgets, or other extended widgets. While extended widgets are normally implemented using portable calls available from basic widgets, basic widgets are highly platform specific, and are normally implemented by the native windowing or operating system. While the `#primaryWidget` message for an extended widget may answer another extended widget, the `#basicWidget` message always returns the basic widget at the end of the chain i.e. the basic widget which represents the extended widget(s) to the native window system.

For CwBasicWidget's, this message returns the receiver itself. For CwExtendedWidgets, this message (recursively) returns the basic widget of the receiver's primary widget.

borderWidth: *anInteger*
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Default: 0 (No Border)
Valid resource values:
  0 (No Border) - Causes the widget to have no border.
  1 (Border) - Causes the widget to have a border.

boundingBox
Answer a rectangle whose origin is the receiver’s x @ y and whose extent is the receiver’s width @ height.

bringToFront
Move the receiver to the top of the stacking order among the receiver’s siblings.

bringToFrontOf: *aCwWidget*
Move the receiver in front of *aCwWidget* in the stacking order among the receiver’s siblings.

children
Answer a collection of the children of the receiver.

configureWidget: *x* **y:** *y* **width:** *width* **height:** *height* **borderWidth:** *borderWidth*
Move and/or resize the receiver, bypassing normal geometry management. This message moves and/or resizes a widget according to the specified width, height, and position values. It returns immediately if the specified geometry fields are the same as the old values. Otherwise, it writes the new x, y, width, height, and borderWidth values into the widget and, if the widget is realized, make the changes visible on the display. A parent widget can use this message to set the geometry of its children. It may also be used to reconfigure a sibling widget.

If only the size of a widget is to be changed, `#resizeWidget:...` is simpler to use; similarly, if only the location of a widget is to be changed, `use #moveWidget:...`

Note that once a widget is resized or otherwise reconfigured by its parent, it may need to do additional processing in its own resize method. Widgets usually need to know when they have changed size so that they can lay out their displayed data again to match the new size. When a parent resizes a child, it calls `#resizeWidget:...`

x,y	The receiver’s new integer x and y coordinates.
width,height,borderWidth	The receiver’s new integer dimensions.

corner
Answer a Point which is the receiver’s (x + width) @ (y + height).

deferRedraw: *aBlock*
Hint that widget display updates for the receiver should be deferred during execution of a block of code. This message is provided as a mechanism to allow the application to provide a hint that the widget updates caused by operations executed in the provided block should be deferred and performed all at once when execution of the block is completed.

The actual effect varies from platform to platform. In general, this should only be used when testing shows it to provide a visual improvement on one or more platforms.

NOTE: Care must be taken not to return out of the middle of aBlock.

destroyWidget
Destroy the receiver and its children and release all associated OS resources. When an application needs to perform additional processing during the destruction of a widget, it should register a destroy callback message for the widget. The destroy callback list is identified by the resource name XmNdestroyCallback. The destroy callback is called just prior to destroying the widget.

display
Answer the CgDisplay associated with the receiver.

disable
Disable the widget.

disableAll
Disable the receiver and all its children, forwarding the `#disable` message to all children. Sending `#disableAll` will result in all children updating their visual appearance to indicate that they are disabled, whereas `#disable` does not.”

dynamicPopupMenu: *aSymbolOrWbMenu* **owner:** *aWbApplication*
Add a dynamic popup menu to the receiver. *aSymbolOrWbMenu* can either be a WbMenu or a symbol that when executed in the context of *aWbApplication* returns a WbMenu. *aSymbol* will be evaluated every time the menu is requested

enable
Enable the widget.

enableAll
Enable the receiver and all its children, forwarding the `#enable` message to all children. Sending `#enableAll` will result in all children updating their visual appearance to indicate that they are enabled, whereas `#enable` does not.”

enabled: *aBoolean*
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

extendedWidget
Answer the extended widget that manages the receiver.

extendedWidgetOrSelf
Answer the extended widget that manages the receiver. If none exists, answer the receiver.

extent
Answer a Point which is the receiver's width @ height.

font: *aFontName*
Set the font of the receiver to *aFontName*.

foregroundColor: *aCgRGBColor*
Specifies the foreground drawing color. **NOTE:** The particular aspects of the widget's appearance which are affected by changing this resource are dependent on platform-specific styles and capabilities and vary from platform to platform.

hasChildren
Answer whether the receiver has children.

hasFocus
Answer true if the receiver has keyboard focus. Otherwise answer false.

height: *anInteger*
Specifies the height of the widget's window in pixels, not including the border area.

hideWindow
Make the receiver invisible.

isVisible
Answer true if the window is visible.

manageChild
Add the receiver to its parent's list of managed children. This message brings a child widget under the geometry management of its parent. A widget cannot be made visible until it is managed.

mapWidget
Map the receiver to its display. This message maps a widget's window to its display, causing it to become visible. A widget must be realized before it can be mapped.

moveWidget: *x* **y:** *y*
Move the receiver on the display. This returns immediately if the specified geometry fields for the widget are the same as the old values. Otherwise, this message writes the new x and y values into the widget and, if the widget is realized, moves the widget on the display.

name
Answer the receiver’s name.

navigationType: *anInteger*
Specifies if tab group navigation is activated for this widget.

Default: XmNONE (None)
Valid resource values:
XmNONE (None) - Indicates that the Widget is not a navigation group
XmTABGROUP (Group) - Indicates that the Widget is included automatically in keyboard navigation

origin
Answer a Point which is the receiver’s x @ y.

owner
Answer the owner of the receiver.

parent
Answer the parent widget of the receiver, or nil if the receiver has no parent.

popupMenu
Answer the popupMenu property (generally a symbol).

popupMenu: *aSymbolOrWbMenu*
Add a popup menu to the receiver. *aSymbolOrWbMenu* can either be a WbMenu or a symbol that when executed in the context of the receiver’s owner returns a WbMenu.

popupMenu: *aSymbolOrWbMenu* **owner:** *aWbApplication*
Add a popup menu to the receiver. *aSymbolOrWbMenu* can either be a WbMenu or a symbol that when executed in the context of *aWbApplication* returns a WbMenu.

primaryWidget
Answer the receiver’s primary widget. The primary widget is any instance of a subclass of CwWidget. For CwBasicWidget’s, the primaryWidget is the receiver itself. For CwExtendedWidgets, the primaryWidget is another CwExtendedWidget, or a CwBasicWidget.

properties
Answer the receiver’s property dictionary.

properties: *anIdentityDictionary*
Set the receiver’s property dictionary.

propertyAt: *aSymbol*
Answer the receiver's property named *aSymbol*.

propertyAt: *aSymbol* **ifAbsent:** *aBlock*
Answer the receiver's property named *aSymbol*. If no such property exists, evaluate *aBlock*.

propertyAt: *aSymbol* **ifAbsentPut:** *aBlock*
Answer the receiver's property named *aSymbol*. If no such property exists, store the value of *aBlock* there.

propertyAt: *aSymbol* **ifMissing:** *anObject*
Answer the receiver's property named *aSymbol*. If no such property exists, store *anObject* there.

propertyAt: *aSymbol* **put:** *anObject*
Set the receiver's property named *aSymbol* to *anObject*.

realChildren
Answer a collection of all of the receivers real children.

realizeWidget
Realize the receiver on the display. This message causes widgets to set create their windows on the display, and perform their final initializations.

realWidget
Answer the receiver's real widget. For most widgets, this is the widget itself. For tightly coupled scrolling widgets, it is the scrolling widget's work window.

redraw
Force the receiver and all children to redraw.

redrawOff
Set redraw off.

redrawOn
Set redraw on.

redraw: *x* **y:** *y* **width:** *width* **height:** *height*
Force the receiver and all children to redraw the specified area of the receiver.

removeAllCallbacks: *callbackName*
Delete all callbacks from a callback list. This message removes all the widget's callback messages identified by callbackName, regardless of the value of the clientData associated with each message. This is in contrast to `#removeCallback:...` and `#removeCallbacks:...`, which remove the specified callback only if a specified clientData argument also matches.

callbackName The resource name of the callback list to which the callback is to be appended.

removeCallback: *callbackName* **receiver:** *receiver* **selector:** *selector* **clientData:** *clientData*
Delete a callback from a callback list. This message removes a callback message identified by callbackName.

The callback is removed only if both the callback object and clientData match a callback/data pair on the list. No warning message is generated if a callback to be removed fails to match a callback or clientData on the list. Use `#removeAllCallbacks:...` if you want to remove a particular callback regardless of the value of its clientData.

callbackName The resource name of the callback list from which the callback is to be deleted.

receiver The object to match against the callback receiver in the callback list.

selector The 3-parameter message selector which is to be used to match against the callback in the callback list.

clientData The object to match with the clientData object in the callback list entry.

removeEventHandler: *eventMask* **receiver:** *receiver* **selector:** *selector* **clientData:** *clientData*
Remove a previously registered event handler. This message stops the specified handler from being called in response to the specified events.

A handler is removed only if both the event handler receiver, selector, and clientData match a previously registered handler/clientData pair.

If a handler to be removed fails to match, or if it has been registered with a different value of clientData, this message returns without reporting an error.

To stop a handler from being called at all, all events for which it is registered must be provided in eventMask. Otherwise, the handler remains registered for the remaining events.

eventMask An integer valued event mask specifying the events of interest. The eventMask parameter is constructed as a bitwise OR of the individual event masks.

receiver The object to match against the event handler receiver in the callback list.

selector The 3-parameter message selector which is to be used to match against the event handler in the callback list.

clientData An object to be passed to the receiver of the event handler message as the clientData parameter when the event handler is invoked, or nil.

resizeWidget: *width* **height:** *height* **borderWidth:** *borderWidth*
Resize a child or sibling widget, bypassing normal geometry management. This message is customarily used by a parent to resize its children.

This message returns immediately if the specified geometry fields are the same as the old values. Otherwise, it writes the new width, height, and borderWidth values into the widget and, if the widget is realized, makes the changes visible.

width,height,borderWidth The receiver's new integer dimensions.

scale
Scale the receiver as needed. This insures that windows laid out under one resolution will look OK under other resolutions.

scaleFactor: *aPoint*
Set the scaling factor to use in laying out the window.

screen
Return the screen for the specified widget. This message returns a CgScreen which describes the screen that the widget is displayed on.

scrolledWidget
Answer the receiver's scrolling widget. For most widgets, this is the widget itself. For tightly coupled scrolling widgets, it is the widget's scrolling parent.

sendToBack
Move the receiver to the bottom of the stacking order among the receiver's siblings.

sendToBackOf: *aCwWidget*
Move the receiver behind *aCwWidget* in the stacking order among the receiver's siblings.

setInputFocus
This message is used to give the receiver the keyboard input focus. If the receiver is a Shell widget, it is activated and brought to the front. If the receiver is not a shell widget, its shell is activated and brought to the front, and then it is given focus.

There are several general rules governing the transfer of keyboard focus:

- A widget can not receive input focus unless its XmNtraversalOn resource is set to true and the XmNtraversalOn resource for all of its ancestors (not including the shell) is set to true.

- An attempt to set focus to a CwComposite widget that has no children will cause the CwComposite widget to take keyboard focus, allowing keyboard events to be received through an event handler.
- An attempt to set focus to a CwComposite widget that has children will cause the CwComposite widget to immediately transfer the keyboard focus to its first child that can receive focus. CwComposite widgets that have children will not normally take keyboard focus.

setSensitive: *sensitive*
Set the sensitivity state of a widget. Many widgets have a mode in which they assume a different appearance (for example, grayed out or stippled), do not respond to user events, and become dormant. When dormant, a widget is insensitive. This means that the widget does not respond to user input events.

setValuesBlock: *argBlock*
This message is provided to allow the values of multiple resources to be set together. Some resources, such as the constraint resources for CwForm are required to be set simultaneously to achieve correct behavior. The argBlock must contain resource set messages to the receiver. The receiver is also passed as the single parameter to the argBlock, and the messages should be sent to this parameter.

shell
Answer the shell widget in which the receiver is contained. Shell widgets answer themselves.

showWindow
Make the receiver visible.

translateCoords: *widgetPoint*
Converts widget-relative coordinates to screen-relative coordinates and answer a Point representing the translated coordinates.

widgetPoint	The Point in widget-relative coordinates which is to be translated to screen-relative coordinates.

traversalOn: *aBoolean*
Specifies if traversal is activated for this widget.

unmanageChild
Remove a widget from its parent’s managed list.

unmapWidget
Unmap a widget explicitly. This message unmaps a widget’s window from its display, causing it to become invisible.

updateDisplay
Synchronize the display by forcing all pending updates for the receiver's display to be processed immediately. All updates for the receiver's display are guaranteed to be processed before the call returns.

NOTE: This can not be called inside an XmNexposeCallback.

updateWidget
Synchronize the display by forcing all pending updates for the receiver to be processed immediately. This may cause pending updates for other widgets to also be processed but at minimum, all updates for the receiver are guaranteed to be processed before the call returns.

NOTE: This can not be called inside an XmNexposeCallback.

visible: *aBoolean*
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

wbCursorPosition
Answer the position of the cursor relative to the receiver.

wbHasFocus
Answer whether the receiver has focus.

wbSetFocusToNext
Set focus to the next widget in the z-order order.

widgetNamed: *aString*
Answer the child of the receiver's named *aString*.

width: *anInteger*
Specifies the width of the widget's window in pixels, not including the border area.

window
Return the CgWindow associated with the receiver.

NOTE: This may return nil if the receiver has not been realized.

withAllChildren
Answer a collection of all of the receiver and the receiver's children.

withAllMajorChildren
Answer a collection of all of the receiver and the receiver's major children.

x: *anInteger*
Specifies the x-coordinate of the widget's upper left-hand corner (excluding the border) in relation to its parent widget.

y: *anInteger*
Specifies the y-coordinate of the widget's upper left-hand corner (excluding the border) in relation to its parent widget.

Callbacks & Events

About To Close Widget
These callbacks are triggered right before the widget is closed.

About To Manage Widget
These callbacks are triggered right after the widget is created but before it is managed.

About To Open Widget
These callbacks are triggered right before the widget is opened.

Button Press
These event handlers are triggered for any mouse button down events.

Button Release
These event handlers are triggered for any mouse button up events.

Button Menu
These event handlers are triggered for any button menu request events.

Button Motion
These event handlers are triggered for any pointer motion events while any button is down.

Button1 Motion
These event handlers are triggered for any pointer motion events while button 1 is down.

Button2 Motion
These event handlers are triggered for any pointer motion events while button 2 is down.

Button3 Motion
These event handlers are triggered for any pointer motion events while button 3 is down.

Closed Widget
These callbacks are triggered right after the widget is closed.

Destroy Callback

These callbacks are triggered when the widget is destroyed. This provides an opportunity to perform any final cleanup activities such as releasing operating system resources.

Help Callback

These callbacks are triggered when the help key sequence is pressed.

Key Press

These event handlers are triggered for any keyboard key down events.

Key Release

These event handlers are triggered for any keyboard key up events.

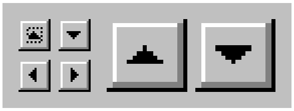
Opened Widget

These callbacks are triggered right after the widget is opened.

Pointer Motion

These event handlers are triggered for all pointer motion events.

CwArrowButton



Arrow buttons are specialized forms of buttons that display an arrow image in one of four directions. Arrow buttons call their activate callback when they are pressed and released.

Protocol

arrowDirection: *anInteger*
Sets the arrow direction.

Default: XmARROWUP (Up Arrow)
Valid resource values:
  XmARROWUP (Up Arrow) - Set the arrow direction to up
  XmARROWDOWN (Down Arrow) - Set the arrow direction to down
  XmARROWLEFT (Left Arrow) - Set the arrow direction to left
  XmARROWRIGHT (Right Arrow) - Set the arrow direction to right

Callbacks & Events

Activate Callback
These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

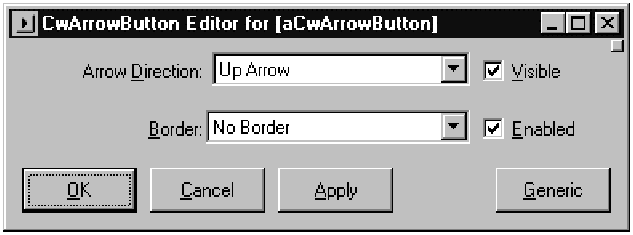
Arm Callback
These callbacks are triggered when the button is armed. Buttons are armed and appear pressed whenever the moused is pressed within the button and not yet released. If the mouse is moved outside of the button while still pressed, the button will be disarmed and appear unpressed. If the mouse is released while still in the button, the button is activated

Disarm Callback
These callbacks are triggered when the button is disarmed. Buttons are disarmed whenever the mouse is moved outside of the button after it has been armed. Moving the mouse back over the button while the mouse button is still down will cause the button to become rearmed.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwArrowButton Editor for [aCwArrowButton]

Arrow Direction: Up Arrow

☑ Visible

Border: No Border

☑ Enabled

Arrow Direction

Sets the arrow direction.

Down Arrow - Set the arrow direction to down
Left Arrow - Set the arrow direction to left
Right Arrow - Set the arrow direction to right
Up Arrow - Set the arrow direction to up

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

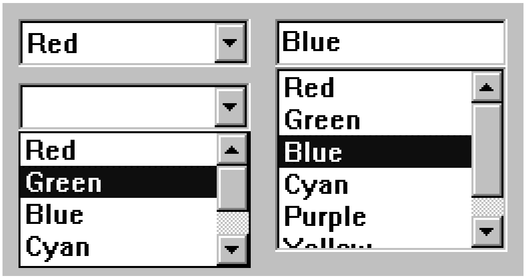
Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwComboBox



Combo box widgets enable the user to select from a list of available items. A combo box also displays the last selected item in a text box above the list. Combo box widgets can only have one item selected at a time.

Protocol

addItem: *item* **position:** *position*
Add an *item* to the list. This message adds an item to the list at the given *position*. A *position* value of 1 makes the first new item the first item in the list; a value of 2 makes it the second item; and so on. A value of 0 makes the first new item follow the last item in the list.

addItems: *items* **position:** *position*
Add *items* to the list. This message adds the specified items to the list at the given *position*. A *position* value of 1 makes the first new item the first item in the list; a value of 2 makes it the second item; and so on. A value of 0 makes the first new item follow the last item in the list.

comboBoxType: *anInteger*
Specifies the style of combo box.

Default: XmDROPDOWN (Drop Down)
Valid resource values:

XmSIMPLE (Simple) - The combo box always displays its list box.

XmDROPDOWN (Drop Down) - the combo box displays its list box only if the user presses the drop down button. When the button is pressed, the list box drops down, allowing the user to make a selection from the list. After the selection is made, the list disappears.

deleteAllItems
This message deletes all items from the list.

deleteItem: *item*
Delete an *item* from the list.

deleteItemsPos: *itemCount* **position:** *position*
Delete items from the list by *position*. This message deletes the specified number of items from the list starting at the specified *position*.

deletePos: *position*
Delete an item from the list by *position*. This message deletes an item at a specified *position*. A warning message appears if the position does not exist.

editable: *aBoolean*
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

getString
This message accesses the String value of the text part of the combo box.

itemCount
Specifies the total number of items in the list. It is automatically updated by the list whenever an element is added to or deleted from the list.

itemExists: *item*
Check if a specified *item* is in the list. This message is a Boolean function that checks if a specified item is present in the list.

items: *anOrderedCollection*
An array of Strings that are to be displayed as the list items.

maxLength: *anInteger*
Specifies the maximum length of the text string that can be entered into text from the keyboard.

replaceItemsPos: *newItems* **position:** *position*
Replace items in the list by position. This message replaces the specified number of items of the List with new items, starting at the specified position in the List. Beginning with the item specified in position, the items in the list are replaced with the corresponding elements from newItems. That is, the item at *position* is replaced with the first element of newItems; the item after *position* is replaced with the second element of newItems; and so on, until *itemCount* is reached.

selectedIndex
Answer the index of the selected item.

selectedItem
Answer the item selected in the combobox.

selectedItems
Answer an array of Strings that represents list items that are currently selected, either by the user or the application.

NOTE: For combo boxes, the collection will contain either 0 or 1 elements.

selectIndex:*itemIndex*
Select the item at *itemIndex*. Index starts at 1.

selectItem: *anObject*
Select the item *anObject*. *anObject* can be an index or a string.

setString: *value*
This message sets the string *value* of the text part of the combo box.

verifyBell: *aBoolean*
Specifies whether the bell should sound when the verification returns without continuing the action.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Activate Callback
These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback
These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

Call data arguments:

text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

Popdown Callback
These callbacks are triggered when the item list disappears

Popup Callback
These callbacks are triggered when the item list appears

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

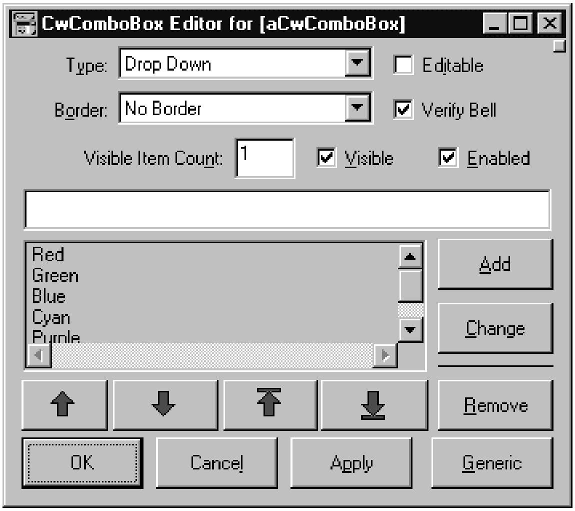
Single Selection Callback
These callbacks are triggered when the user selects an item in the list, or presses an arrow key to move through the list.

Call data arguments:

item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.

Value Changed Callback
These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



CwComboBox Editor for [aCwComboBox]

Type: Drop Down ☐ Editable

Border: No Border ☑ Verify Bell

Visible Item Count: 1 ☑ Visible ☑ Enabled

Red
Green
Blue
Cyan
Purple

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Combo Box Type

Specifies the style of combo box.

Drop Down - the combo box displays its list box only if the user presses the drop down button. When the button is pressed, the list box drops down, allowing the user to make a selection from the list. After the selection is made, the list disappears.
Simple - The combo box always displays its list box.

Editable

Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Items

An array of Strings that are to be displayed as the list items.

Max Length

Specifies the maximum length of the text string that can be entered into text from the keyboard.

Verify Bell

Specifies whether the bell should sound when the verification returns without continuing the action.

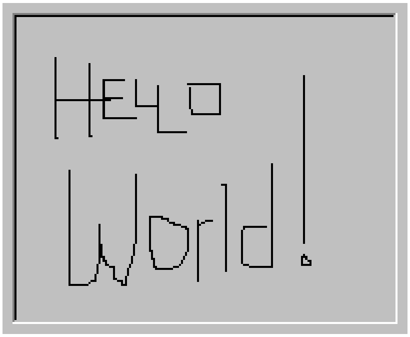
Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Visible Item Count

Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

CwDrawingArea



The drawing area widget provides an application with an area in which application-defined graphics can be drawn using Common Graphics operations such as `#fillRectangle:`, `#drawArc:`, and `#drawString:`. Consult the Common Graphics chapter of the *VisualAge Programmer's Reference* for an explanation of drawing and other graphics operations.

Drawing is actually done on the CgWindow associated with the CwDrawingArea. Every CwWidget has a corresponding CgWindow, obtained with aCwWidget window, that can be used for drawing. Although any widget can be drawn on in this manner, CwDrawingArea widgets are typically used since they provide additional drawing-related functionality.

A CwDrawingArea can be told to notify the application with an expose callback whenever a part of the drawing area needs to be redrawn. The expose callback contains an expose event with a rectangle describing the damaged area of the widget's CgWindow.

Protocol

marginHeight: *anInteger*
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

marginWidth: *anInteger*
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

resizePolicy: *anInteger*
Specifies the resize policy of the widget.

Default: XmRESIZEANY (Any)
Valid resource values:
XmRESIZENONE (None) - Resize none.
XmRESIZEGROW (Grow) - Resize grow.
XmRESIZEANY (Any) - Resize any.

Callbacks & Events

Expose Callback
These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Call data arguments:
event - the CwEvent associated with the receiver.
window - the widget's CgWindow which can be used for drawing purposes.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Input Callback
These callbacks are triggered when the widget processes a keyboard or mouse event (key or button, up or down).

Call data arguments:
event - the CwEvent associated with the receiver.
window - the widget's CgWindow which can be used for drawing purposes.

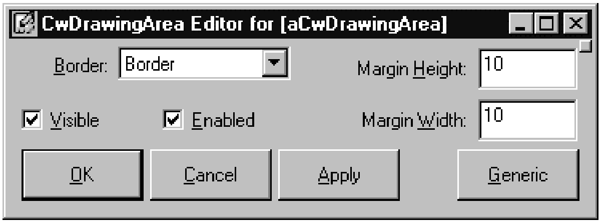
Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Call data arguments:
event - the CwEvent associated with the receiver.
window - the widget's CgWindow which can be used for drawing purposes.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwDrawingArea Editor for [aCwDrawingArea]

Border: Border

Margin Height: 10

☑ Visible ☑ Enabled

Margin Width: 10

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

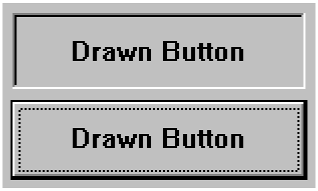
Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Margin Height
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

Margin Width
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwDrawnButton



Application-drawn button widgets enable the application to draw arbitrary graphics on a button. Drawn buttons behave like push buttons except that they can be drawn on like drawing area widgets.

As with the push button widget, the application can add an activate callback to be executed when the button is pressed. As with the drawing area widget, expose and resize callbacks can be added to notify the application when the button requires redrawing and when it has changed size.

Protocol

alignment: *anInteger*
Specifies the label alignment for text or pixmap.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the left sides of the lines of text to be vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
XmALIGNMENTCENTER (Center) - Causes the centers of the lines of text to be vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window
XmALIGNMENTEND (Right) - Causes the right sides of the lines of text to be vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

click
Programatically click the button.

labelInsensitivePixmap: *aCgPixmap*
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

labelPixmap: *aCgPixmap*
Specifies the pixmap when label type is Pixmap.

labelString: *aString*
Specifies the label string when the label type is String.

labelType: *anInteger*
Specifies the label type.

Default: XmSTRING (Text)
Valid resource values:
XmPIXMAP (Pixmap) - Causes the label to display a pixmap
XmSTRING (Text) - Causes the label to display text
XmICON (Icon) - Causes the label to display an icon

mnemonic: *aCharacter*
Provides the user with alternate means for selecting a button.

pushButtonEnabled: *aBoolean*
Enables or disables the three-dimensional shadow drawing as in PushButton.

recomputeSize: *aBoolean*
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Callbacks & Events

Activate Callback

These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Call data arguments:
event - the CwEvent associated with the receiver.
window - the widget's CgWindow which can be used for drawing purposes.

Arm Callback

These callbacks are triggered when the button is armed. Buttons are armed and appear pressed whenever the moused is pressed within the button and not yet released. If the mouse is moved outside of the button while still pressed, the button will be disarmed and appear unpressed. If the mouse is released while still in the button, the button is activated

Disarm Callback

These callbacks are triggered when the button is disarmed. Buttons are disarmed whenever the mouse is moved outside of the button after it has been armed. Moving the mouse back over the button while the mouse button is still down will cause the button to become rearmed.

Expose Callback

These callbacks are triggered when the button receives an exposure event requiring it to repaint itself.

Call data arguments:

event - the CwEvent associated with the receiver.
window - the widget's CgWindow which can be used for drawing purposes.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

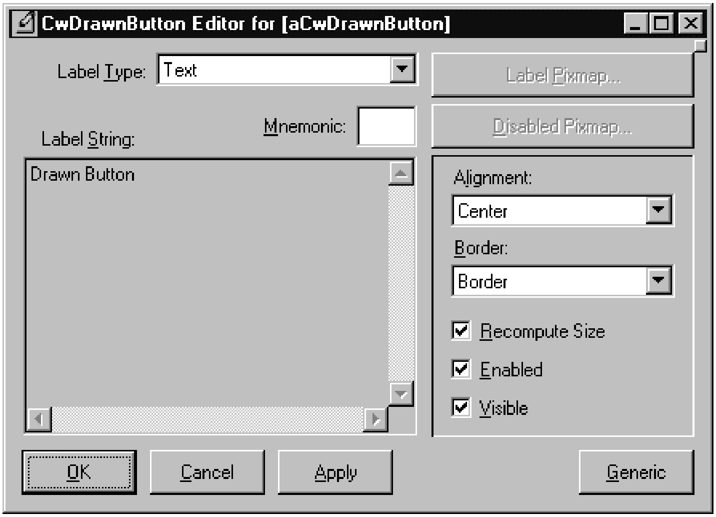
Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback

These callbacks are triggered when the button receives a resize event. This allows the button to perform any calculation to adjust the size of the image that it displays.

Editor



CwDrawnButton Editor for [aCwDrawnButton]

Label Type: Text

Mnemonic:

Label String:

Drawn Button

Alignment: Center

Border: Border

☑ Recompute Size

☑ Enabled

☑ Visible

Alignment
Specifies the label alignment for text or pixmap.

Center - Causes the centers of the lines of text to be vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window
Left - Causes the left sides of the lines of text to be vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
Right - Causes the right sides of the lines of text to be vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Label Insensitive Pixmap
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

Label Pixmap
Specifies the pixmap when label type is Pixmap.

Label String
Specifies the label string when the label type is String.

Label Type
Specifies the label type.

Icon - Causes the label to display an icon
Pixmap - Causes the label to display a pixmap
Text - Causes the label to display text

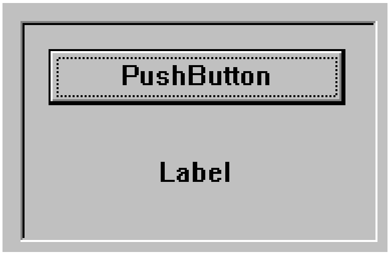
Mnemonic
Provides the user with alternate means for selecting a button.

Push Button Enabled
Enables or disables the three-dimensional shadow drawing as in PushButton.

Recompute Size
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwForm



Form widgets are composite widgets that allow the application to specify how child widgets of the composite should be laid out relative to each other and relative to the composite. Form widget children are positioned by attaching their sides to other objects. Attachments are specified by setting each child’s leftAttachment, rightAttachment, topAttachment and bottomAttachment resources. A side can be attached either to a given position, to another widget, or to the edge of the form. The attachment types are listed below. The first four types are the most commonly used. All are described in terms of the leftAttachment, but the same attachment types apply to the other sides, with corresponding behavior.

Protocol

fractionBase: *anInteger*
Specifies the denominator used in calculating the relative position of the child widgets.

horizontalSpacing: *anInteger*
Specifies the offset for right and left attachments.

marginHeight: *anInteger*
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

marginWidth: *anInteger*
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

resizePolicy: *anInteger*
Specifies the resize policy of the widget.

Default: XmRESIZEANY (Any)
Valid resource values:
XmRESIZENONE (None) - Resize none.
XmRESIZEGROW (Grow) - Resize grow.
XmRESIZEANY (Any) - Resize any.

rubberPositioning: *aBoolean*
Indicates the default attachment for a child of the Form. If this Boolean resource is set to false, then the left and top of the child defaults to being attached to the left and top side of the Form. If this resource is set to true, then the child defaults to being attached to its initial position in the Form.

verticalSpacing: *anInteger*
Specifies the offset for top and bottom attachments.

Callbacks & Events

Expose Callback
These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

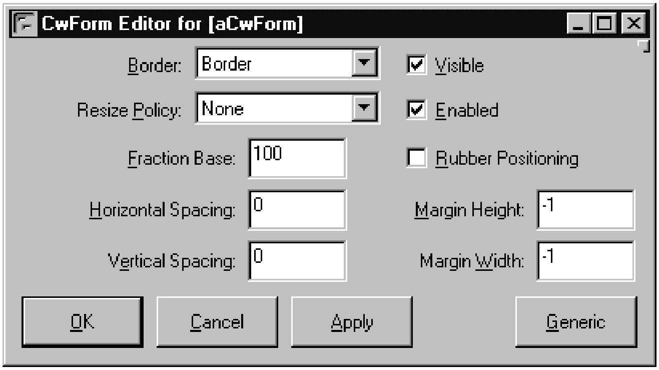
Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwForm Editor for [aCwForm]

Border: Border

Resize Policy: None

Fraction Base: 100

Horizontal Spacing: 0

Vertical Spacing: 0

☑ Visible

☑ Enabled

☐ Rubber Positioning

Margin Height: -1

Margin Width: -1

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Fraction Base
Specifies the denominator used in calculating the relative position of the child widgets.

Horizontal Spacing
Specifies the offset for right and left attachments.

Margin Height
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

Margin Width
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

Resize Policy
Specifies the resize policy of the widget.

Any - Resize any.
Grow - Resize grow.
None - Resize none.

Rubber Positioning

Indicates the default attachment for a child of the Form. If this Boolean resource is set to false, then the left and top of the child defaults to being attached to the left and top side of the Form. If this resource is set to true, then the child defaults to being attached to its initial position in the Form.

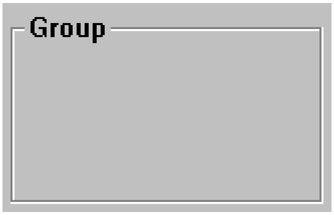
Vertical Spacing

Specifies the offset for top and bottom attachments.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwFrame



Frame widgets are used to visually indicate and label groups of related controls. They are composed of a box with an optional label in the upper left corner.

***Note:** Frame widgets in VisualAge Smalltalk are allowed to have one and only one child. In order for a frame to group multiple widgets, a form widget should be inserted into the frame as its sole child. The form then acts as the parent of any other widgets placed within the bounds of the frame. The **Always Add Forms To Frames** command will cause a form to be automatically inserted into any new frame.*

Protocol

labelString: *aString*
Specifies the label string.

marginHeight: *anInteger*
Specifies the padding space on the top and bottom sides between the child of Frame and Frame's shadow drawing.

marginWidth: *anInteger*
Specifies the padding space on the left and right sides between the child of Frame and Frame's shadow drawing.

shadowType: *anInteger*
Describes the drawing style for Frame.

Default: XmSHADOWDEFAULT (Default)
Valid resource values:
XmSHADOWDEFAULT (Default) - Draws Frame in a platform specific manner.
XmSHADOWETCHEDIN (Etched In) - Draws Frame using a double line giving the effect of a line etched into the window.
XmSHADOWETCHEDOUT (Etched Out) - Draws Frame using a double line giving the effect of a line coming out of the window.

XmSHADOWIN (In) - Draws Frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
XmSHADOWOUT (Out) - Draws Frame such that it appears outset.

Callbacks & Events

Expose Callback

These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback

These callbacks are triggered when any area of the widget or its children is exposed.

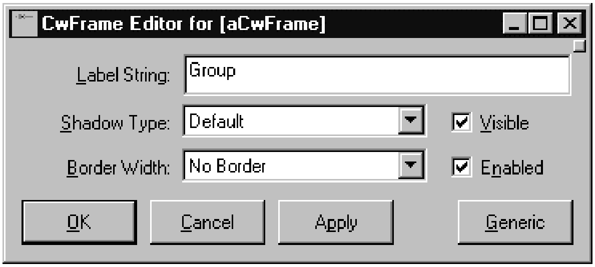
Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwFrame Editor for [aCwFrame]

Label String: Group
Shadow Type: Default
☑ Visible
Border Width: No Border
☑ Enabled

Border Width

Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

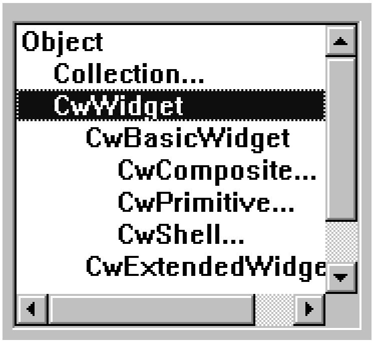
Label String
Specifies the label string.

Shadow Type
Describes the drawing style for Frame.

Default - Draws Frame in a platform specific manner.
Etched In - Draws Frame using a double line giving the effect of a line etched into the window.
Etched Out - Draws Frame using a double line giving the effect of a line coming out of the window.
In - Draws Frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
Out - Draws Frame such that it appears outset.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwHierarchyList



Hierarchy lists extend the capabilities of a normal list box to view a hierarchy of objects, rather than simply a flat list of strings. With them you can view any group of objects that have a hierarchical ordering. Objects that are children if other objects will be so indicated with indentation, and parent objects can be expanded/collapsed to reveal/hide their children.

Protocol

addAllShowing: *aCollection*
Add the items in aCollection to the set of objects which are showing their children.

addItem: *item* **position:** *position*
Add an *item* to the list. This message adds an item to the list at the given *position*. When the item is inserted into the list, it is compared with the current selectedItems list. If the new item matches an item on the selected list, it appears selected.

addItems: *items* **position:** *position*
Add *items* to the list. This message adds the specified items to the list at the given *position*. When the items are inserted into the list, they are compared with the current selectedItems list. If any of the new items matches an item on the selected list, it appears selected.

addItemUnselected: *item* **position:** *position*
Add an *item* to the list, forcing it to be unselected. This message adds an item to the list at the given *position*. The item does not appear selected, even if it matches an item in the current selectedItems list.

addShowing: *anItem*
Add *anItem* to the collection of objects which are showing their children.

childrenSelector: *aSymbol*
Specify the selector which determines the children of each child of the list. The selector should expect a parameter (a list item).

deleteAllItems
This message deletes all items from the list.

deleteItem: *item*
Delete an *item* from the list.

deleteItems: *items*
This message deletes the specified *items* from the list. A warning message appears if any of the items do not exist.

deleteItemsPos: *itemCount* **position:** *position*
Delete items from the list by position. This message deletes the specified number of items from the list starting at the specified *position*.

deletePos: *position*
Delete an item from the list by *position*. This message deletes an item at a specified position. A warning message appears if the position does not exist.

deselectAllItems
Unhighlight and remove all elements from the selectedItems list.

deselectItem: *item*
Unhighlight and remove the specified *item* from the selected list.

deselectPos: *position*
Unhighlight and remove an item from the selected list by *position*.

getMatchPos: *item*
This message returns an Array of Integer positions where a specified *item* is found in a List. If the item does not occur in the list the resulting Array is empty. The #= operator is used for the search.

getSelectedPos
Return an Array containing the positions of every selected item in the list.

hasChildrenSelector: *aSymbol*
Specify the selector which the model uses to determine whether an item in the list has children. The selector is expected to return a Boolean. Note that it is the model's responsibility to ensure that if this selector is used, that it returns a value consistent with that returned by the `#childrenSelector`. i.e. If the `#hasChildrenSelector` returns false, then the `#childrenSelector` should not return a non-empty collection.

hideShow
Toggle whether the objects in aCollection should have their children shown in the pane. Objects which are to be hidden must *not* have all of their descendants removed from the showing list as well. This means that if the contents of the object are shown again, the entire previously shown hierarchy will be shown, without requiring the user to re-select the items to show. Refresh the display list and restore the selected items.

hideShowAll
Toggle whether the objects in aCollection should have their descendants shown in the pane. Objects which are to be hidden will have all of their descendants removed from the showing list as well. Objects which are to be shown will have all of their descendants added to the showing list. Refresh the display list and the restore the currently selected items.

hierarchySelector: *aSymbol*
Specify the selector which the model uses to set the initial hierarchy dictionary of the receiver. The selector is expected to return an IdentityDictionary which maps the items in the list to their contents. Note that the hierarchy does NOT have to include all of the possible elements in the list. Any descendants from the roots which have are not included in the dictionary will be added as required using the `#membersSelector`.

isShowing: *anItem*
Return whether the receiver has anItem as one of the items which are showing their descendants in the pane.

itemCount
Answer the total number of items. It is automatically updated by the list whenever an element is added to or deleted from the list.

itemExists: *item*
Check if a specified *item* is in the list. This message is a Boolean function that checks if a specified item is present in the list. The #= operator is used for the search.

listMsg: *aSymbol*
Specifies the selector that is sent to the model to return the initial list.

parentSelector: *aSymbol*
Specify the selector which determines parent of a list item. Used (among other things) to determine the indentation of the items in the list. The selector should expect a parameter (a list item).

printItems: *aCollection*
Answer the printable string representations for the *aCollection* to be used as the items collection for the primary widget.

printSelector: *aSymbol*
Specifies the selector that is used to obtain printable string representations for the items by evaluating it with each item.

removeShowing: *anItem*
Remove *anItem* from the collection of objects which are showing their children. Do nothing if *anItem* is missing from the collection.

removeShowing: *anItem* **ifAbsent:** *aBlock*
Remove *anItem* from the collection of objects which are showing their children. Return the value of *aBlock* if *anItem* is missing from the collection.

replaceItems: *oldItems* **newItems:** *newItems*
This message replaces each specified item of the list with a corresponding new item. Every occurrence of each element of oldItems is replaced with the corresponding element from newItems. That is, the first element of oldItems is replaced with the first element of newItems. The second element of oldItems is replaced with the second element of newItems, and so on. The #= operator is used for the search.

replaceItemsPos: *newItems* **position:** *position*
Replace items in the list by position. This message replaces the specified number of items of the List with new items, starting at the specified position in the List. Beginning with the item specified in *position*, the items in the list are replaced with the corresponding elements from *newItems*. That is, the item at *position* is replaced with the first element of *newItems*; the item after *position* is replaced with the second element of *newItems*; and so on, until itemCount is reached.

scrollHorizontal: *aBoolean*
This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

selectedItems: *aCollection*
Set the array of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:

XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

selectItem: *item* **notify:** *notify*
Select an *item* in the list. This message highlights and adds the specified *item* to the current selected list. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

selectPos: *position* **notify:** *notify*
Select an item in the list by *position*. This message highlights a List item at the specified *position* and adds it to the list of selected items. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

setBottomItem: *item*
Make an existing *item* the last visible item in the list. This message makes an existing item the last visible item in the list. The item can be any valid item in the list.

setBottomPos: *position*
Make an item the last visible item in the list by *position*. This message makes the item at the specified *position* the last visible item in the List.

setItem: *item*
Make an existing *item* the first visible item in the list. This message makes an existing item the first visible item in the list. The item can be any valid item in the list.

setPos: *position*
Make an item the first visible item in the list by *position*. This message makes the item at the given *position* the first visible position in the List.

showing: *aCollection*
Set the collection of objects which are showing their children to be *aCollection*.

topItemPosition: *anInteger*
Specifies the position of the item that is the first visible item in the list.

update
Update the receiver list and the items displayed.

updateItems
Update the display using the collection of items that have been determined that need to be displayed. It is assumed that itemList contains this list. items is a collection relating to the underlying primary widget.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Browse Selection Callback
These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Default Action Callback
These callbacks are triggered when an item is double clicked.

Extended Selection Callback
These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Modify Verify Callback
Specifies a list of callbacks that is called when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

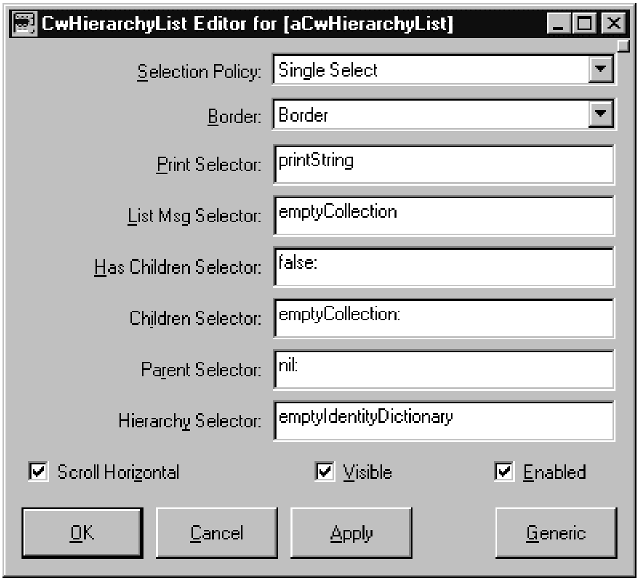
Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback
These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Single Selection Callback
These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Editor



CwHierarchyList Editor for [aCwHierarchyList]

Selection Policy: Single Select

Border: Border

Print Selector: printString

List Msg Selector: emptyCollection

Has Children Selector: false:

Children Selector: emptyCollection:

Parent Selector: nil:

Hierarchy Selector: emptyIdentityDictionary

☑ Scroll Horizontal ☑ Visible ☑ Enabled

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Children Selector
Specify the selector which determines the children of each child of the list. The selector should expect a parameter (a list item).

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Has Children Selector
Specify the selector which the model uses to determine whether an item in the list has children. The selector is expected to return a Boolean. Note that it is the model’s responsibility to ensure that if this selector is used, that it returns a value consistent with that returned by the `#childrenSelector`. i.e. If the `#hasChildrenSelector` returns false, then the `#childrenSelector` should not return a non-empty collection.

Hierarchy Selector
Specify the selector which the model uses to set the initial hierarchy dictionary of the receiver. The selector is expected to return an IdentityDictionary which maps the items in the list to their contents. Note that the hierarchy does NOT have to include all of the possible elements in the list. Any descendants from the roots which have are not included in the dictionary will be added as required using the #membersSelector.

List Msg Selector
Specifies the selector that is sent to the model to return the initial list.

Parent Selector
Specify the selector which determines parent of a list item. Used (among other things) to determine the indentation of the items in the list. The selector should expect a parameter (a list item).

Print Selector
Specifies the selector that is used to obtain printable string representations for the items by evaluating it with each item.

Scroll Horizontal
This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

Selection Policy
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

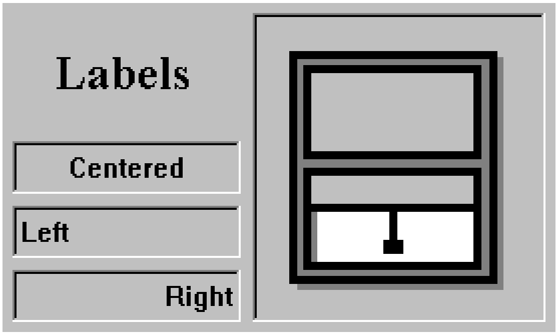
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

Top Item Position
Specifies the position of the item that is the first visible item in the list.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwLabel



Labels are static controls that can display either strings, pixmaps or icons as their contents, depending on the value of the labelType resource. Static labels do not provide any special callbacks.

Protocol

alignment: *anInteger*
Specifies the label alignment for text or pixmap.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:

XmALIGNMENTBEGINNING (Left) - Causes the left sides of the lines of text to be vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.

XmALIGNMENTCENTER (Center) - Causes the centers of the lines of text to be vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window

XmALIGNMENTEND (Right) - Causes the right sides of the lines of text to be vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

labelInsensitivePixmap: *aCgPixmap*
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

labelPixmap: *aCgPixmap*
Specifies the pixmap when label type is Pixmap.

labelString: *aString*
Specifies the label string when the label type is String.

labelType: *anInteger*
Specifies the label type.

Default: XmSTRING (Text)
Valid resource values:
XmPIXMAP (Pixmap) - Causes the label to display a pixmap
XmSTRING (Text) - Causes the label to display text
XmICON (Icon) - Causes the label to display an icon

mnemonic: *aCharacter*
Provides the user with alternate means for selecting a button.

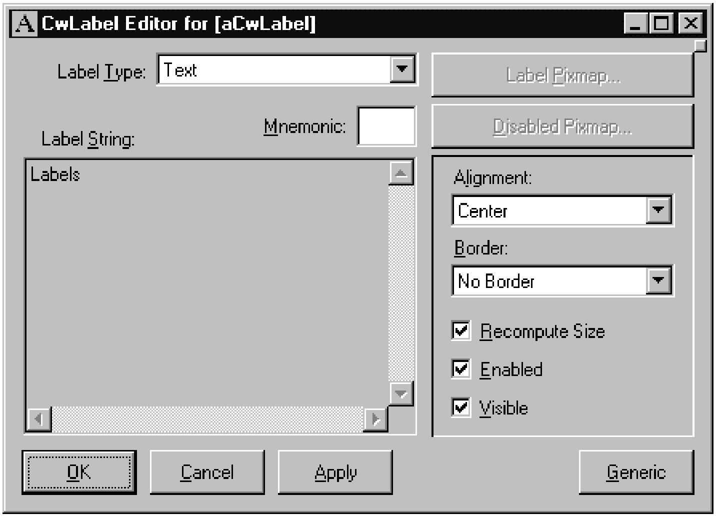
recomputeSize: *aBoolean*
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Callbacks & Events

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwLabel Editor for [aCwLabel]

Label Type: Text

Label String: Labels

Mnemonic:

Alignment: Center

Border: No Border

☑ Recompute Size

☑ Enabled

☑ Visible

Alignment

Specifies the label alignment for text or pixmap.

Center - Causes the centers of the lines of text to be vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window
Left - Causes the left sides of the lines of text to be vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
Right - Causes the right sides of the lines of text to be vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Label Insensitive Pixmap

Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

Label Pixmap

Specifies the pixmap when label type is Pixmap.

Label String

Specifies the label string when the label type is String.

Label Type

Specifies the label type.

Icon - Causes the label to display an icon
Pixmap - Causes the label to display a pixmap
Text - Causes the label to display text

Mnemonic

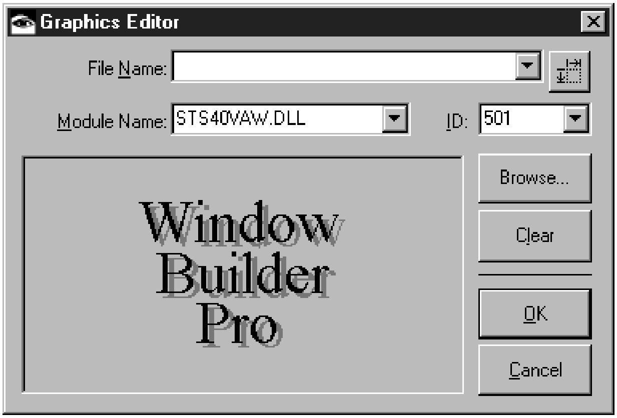
Provides the user with alternate means for selecting a button.

Recompute Size

Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Graphics Editor



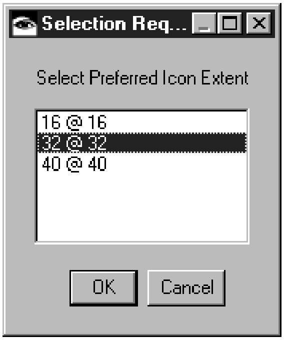
File Name:

Module Name: STS40VAW.DLL

ID: 501

File Name
Specifies the file name of the graphic file to be used as the label for the widget. A variety of graphics file formats are supported including standard bitmaps (BMP files), icons (ICO files), PCX file and TIFF files. If the file name does not include a path, the file is assumed to reside in the local directory or in the system bitmaps directory.

Preferred Icon Extent
Specifies preferred icon extent which is used in the event that the icon file contains multiple icon resources of different sizes.



Select Preferred Icon Extent

16 @ 16
32 @ 32
40 @ 40

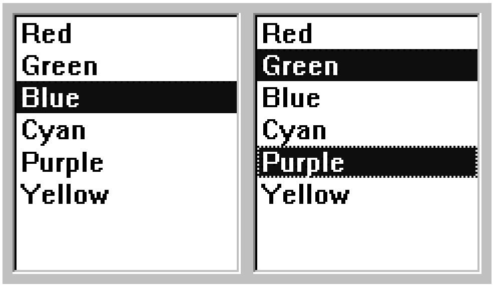
Module Name
Specifies the name of the module (DLL) containing the bitmap or icon to be used as the label for the widget

ID
Specifies the ID of the of the bitmap in the specified module.

Browse
Opens a file dialog from which a graphics file may be selected.

Clear
Clears any graphic choice that has been made.

CwObjectList



Object lists extend the capabilities of a normal list box to view a collection of objects, rather than simply a flat list of strings.

Protocol

addItem: *item* **position:** *position*
Add an *item* to the list. This message adds an item to the list at the given *position*. When the item is inserted into the list, it is compared with the current selectedItems list. If the new item matches an item on the selected list, it appears selected.

addItems: *items* **position:** *position*
Add *items* to the list. This message adds the specified items to the list at the given *position*. When the items are inserted into the list, they are compared with the current selectedItems list. If the any of the new items matches an item on the selected list, it appears selected.

addItemUnselected: *item* **position:** *position*
Add an *item* to the list, forcing it to be unselected. This message adds an *item* to the list at the given *position*. The item does not appear selected, even if it matches an item in the current selectedItems list.

deleteAllItems
This message deletes all items from the list.

deleteItem: *item*
Delete an *item* from the list.

deleteItems: *items*
This message deletes the specified *items* from the list. A warning message appears if any of the items do not exist.

deleteItemsPos: *itemCount* **position:** *position*
Delete items from the list by *position*. This message deletes the specified number of items from the list starting at the specified position.

deletePos: *position*
Delete an item from the list by *position*. This message deletes an item at a specified position. A warning message appears if the position does not exist.

deselectAllItems
Unhighlight and remove all elements from the selectedItems list.

deselectItem: *item*
Unhighlight and remove the specified *item* from the selected list.

deselectPos: *position*
Unhighlight and remove an item from the selected list by *position*.

getMatchPos: *item*
This message returns an Array of Integer positions where a specified *item* is found in a List. If the item does not occur in the list the resulting Array is empty. The #= operator is used for the search.

getSelectedPos
Return an Array containing the positions of every selected item in the list.

itemCount
Answer the total number of items. It is automatically updated by the list whenever an element is added to or deleted from the list.

itemExists: *item*
Check if a specified *item* is in the list. This message is a Boolean function that checks if a specified item is present in the list. The #= operator is used for the search.

items: *anOrderedCollection*
An array of Strings that are to be displayed as the list items.

printSelector: *aSymbol*
Specifies the selector that is used to obtain printable string representations for the items by evaluating it with each item.

replaceItems: *oldItems* **newItems:** *newItems*
This message replaces each specified item of the list with a corresponding new item. Every occurrence of each element of *oldItems* is replaced with the corresponding element from *newItems*. That is, the first element of *oldItems* is replaced with the first element of

newItems. The second element of *oldItems* is replaced with the second element of *newItems*, and so on. The #= operator is used for the search.

replaceItemsPos: *newItems* **position:** *position*
Replace items in the list by position. This message replaces the specified number of items of the List with new items, starting at the specified *position* in the List. Beginning with the item specified in *position*, the items in the list are replaced with the corresponding elements from *newItems*. That is, the item at *position* is replaced with the first element of *newItems*; the item after *position* is replaced with the second element of *newItems*; and so on, until itemCount is reached.

scrollHorizontal: *aBoolean*
This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

selectedItemCount
Answer the number of strings in the selected items list.

selectedItems: *anOrderedCollection*
An array of Strings that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:

XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

selectItem: *item* **notify:** *notify*
Select an *item* in the list. This message highlights and adds the specified *item* to the current selected list. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

selectPos: *position* **notify:** *notify*
Select an item in the list by *position*. This message highlights a List item at the specified *position* and adds it to the list of selected items. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

setBottomItem: *item*
Make an existing *item* the last visible item in the list. This message makes an existing item the last visible item in the list. The item can be any valid item in the list.

setBottomPos: *position*
Make an item the last visible item in the list by *position*. This message makes the item at the specified *position* the last visible item in the List.

setItem: *item*
Make an existing *item* the first visible item in the list. This message makes an existing item the first visible item in the list. The item can be any valid item in the list.

setPos: *position*
Make an item the first visible item in the list by *position*. This message makes the item at the given *position* the first visible position in the List.

topItemPosition: *anInteger*
Specifies the position of the item that is the first visible item in the list.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

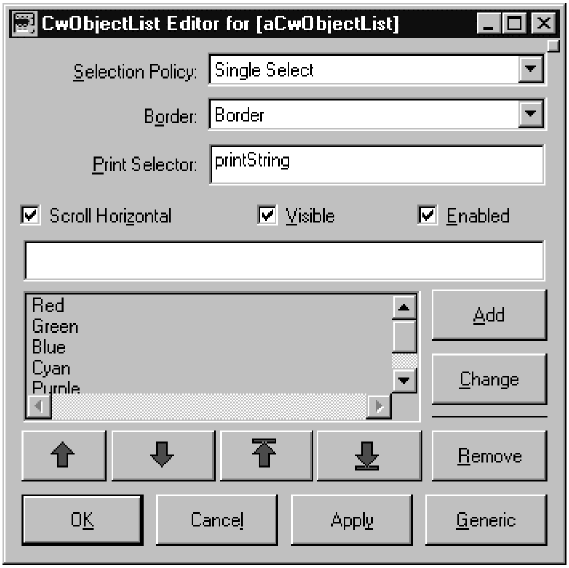
Modify Verify Callback
Specifies a list of callbacks that is called when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

Call data arguments:
doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback
These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Single Selection Callback
These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Editor



CwObjectList Editor for [aCwObjectList]

Selection Policy: Single Select

Border: Border

Print Selector: printString

☑ Scroll Horizontal ☑ Visible ☑ Enabled

Red
Green
Blue
Cyan
Purple

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Items

An array of Strings that are to be displayed as the list items.

Print Selector

Specifies the selector that is used to obtain printable string representations for the items by evaluating it.

Scroll Horizontal

This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

Selected Items

An array of Strings that represents the list items that are currently selected, either by the user or the application.

Selection Policy

Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

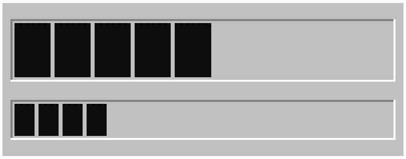
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwProgressBar



CwProgressBar is a visual control which is used to show the progress of a time consuming operation by the gradual filling from left to right of a long flat rectangle. You can use it to give a user of your application an indication, time-wise, of how an operation is progressing. A progress bar has minimum, maximum, and value properties. The minimum and maximum properties define the range of an operation, the operation on which you wish to report progress. The value property defines a point within the range and by doing so, specifies the operation's progress. For example, a progress bar with minimum of 0, maximum of 10, and value of 5, indicates an operation that is 50% complete. You convey status to a user of your application during a time consuming operation by incrementing, decrementing, or setting the value of the value attribute. This causes appropriate changes in the filling of the progress bar. This widget is only available in the Windows version of VisualAge.

Protocol

maximum: *anInteger*
Specifies the maximum value of the progress bar.

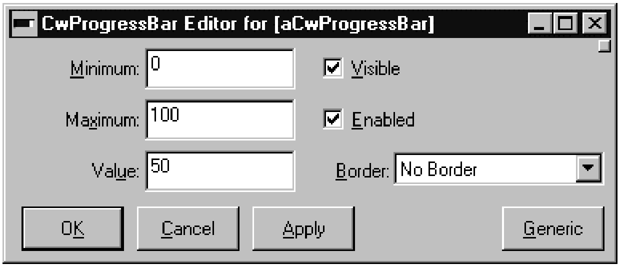
minimum: *anInteger*
Specifies the minimum value of the progress bar.

value: *anInteger*
Specifies the current position of the progress bar.

Callbacks & Events

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwProgressBar Editor for [aCwProgressBar]

Minimum: 0

Maximum: 100

Value: 50

☑ Visible

☑ Enabled

Border: No Border

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

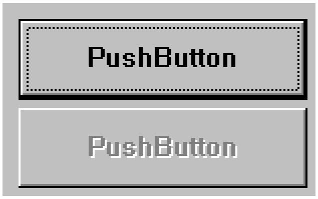
Maximum
Specifies the slider maximum value.

Minimum
Specifies the slider minimum value.

Value
Specifies the slider current position along the scale, between minimum and maximum.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwPushButton



Push buttons provide a mechanism to initiate an action when clicked on. They call their activate callback when they are pressed and released. Buttons can display either strings, pixmaps or icons as their contents, depending on the value of the labelType resource.

Protocol

click
Programatically click the button.

defaultButton: *aBoolean*
Set the receiver to be the default button

labelInsensitivePixmap: *aCgPixmap*
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

labelPixmap: *aCgPixmap*
Specifies the pixmap when label type is Pixmap.

labelString: *aString*
Specifies the label string when the label type is String.

labelType: *anInteger*
Specifies the label type.

Default: XmSTRING (Text)
Valid resource values:
XmPIXMAP (Pixmap) - Causes the label to display a pixmap
XmSTRING (Text) - Causes the label to display text
XmICON (Icon) - Causes the label to display an icon

mnemonic: *aCharacter*
Provides the user with alternate means for selecting a button.

recomputeSize: *aBoolean*
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

showAsDefault: *anInteger*
Specifies a shadow thickness for a second shadow to be drawn around the PushButton to visually mark it as a default button. When the resource is set to 0, the button appears as a normal button.

Default: 0 (Normal)
Valid resource values:
    0 (Normal) - Causes the PushButton to look like a normal button
    1 (Default) - Causes the PushButton to look like a default button

Callbacks & Events

Activate Callback

These callbacks are triggered when the button has been activated. Buttons are activated when the mouse is clicked and released within the button. Buttons may also be activated via the space bar when the button has focus or via a carriage return when a button is a default button.

Arm Callback

These callbacks are triggered when the button is armed. Buttons are armed and appear pressed whenever the mouse is pressed within the button and not yet released. If the mouse is moved outside of the button while still pressed, the button will be disarmed and appear unpressed. If the mouse is released while still in the button, the button is activated.

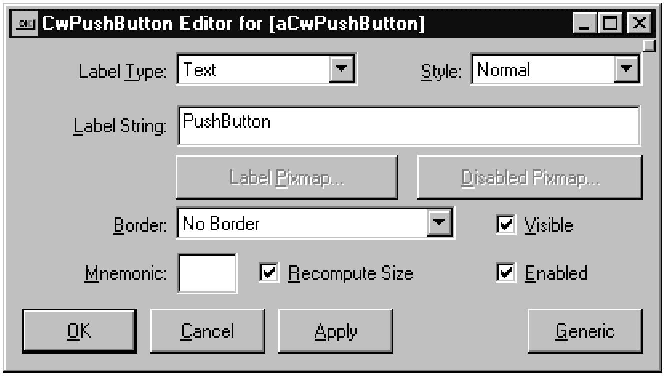
Disarm Callback

These callbacks are triggered when the button is disarmed. Buttons are disarmed whenever the mouse is moved outside of the button after it has been armed. Moving the mouse back over the button while the mouse button is still down will cause the button to become rearmed.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwPushButton Editor for [aCwPushButton]

Label Type: Text

Style: Normal

Label String: PushButton

Border: No Border

☑ Visible

Mnemonic:

☑ Recompute Size

☑ Enabled

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Label Insensitive Pixmap
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

Label Pixmap
Specifies the pixmap when label type is Pixmap.

Label String
Specifies the label string when the label type is String.

Label Type
Specifies the label type.

Icon - Causes the label to display an icon
Pixmap - Causes the label to display a pixmap
Text - Causes the label to display text

Mnemonic
Provides the user with alternate means for selecting a button.

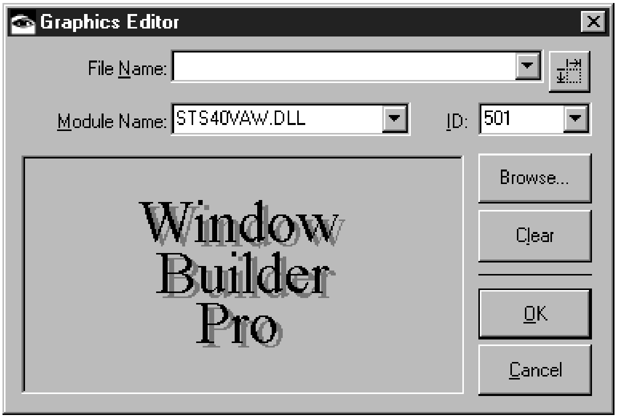
Recompute Size
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Show As Default
Specifies a shadow thickness for a second shadow to be drawn around the PushButton to visually mark it as a default button. When the resource is set to 0, the button appears as a normal button.

> Default - Causes the PushButton to look like a default button
> Normal - Causes the PushButton to look like a normal button

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

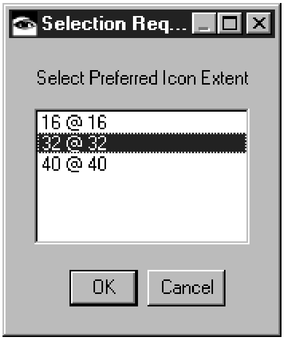
Graphics Editor



File Name:
Module Name: STS40VAW.DLL
ID: 501

File Name
Specifies the file name of the graphic file to be used as the label for the widget. A variety of graphics file formats are supported including standard bitmaps (BMP files), icons (ICO files), PCX file and TIFF files. If the file name does not include a path, the file is assumed to reside in the local directory or in the system bitmaps directory.

Preferred Icon Extent
Specifies preferred icon extent which is used in the event that the icon file contains multiple icon resources of different sizes.



Select Preferred Icon Extent

16 @ 16
32 @ 32
40 @ 40

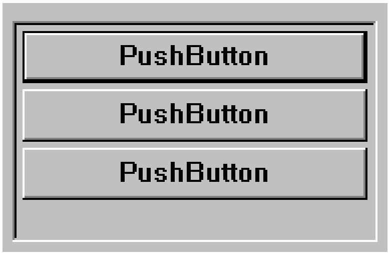
Module Name
Specifies the name of the module (DLL) containing the bitmap or icon to be used as the label for the widget

ID
Specifies the ID of the of the bitmap in the specified module.

Browse
Opens a file dialog from which a graphics file may be selected.

Clear
Clears any graphic choice that has been made.

CwRowColumn



Row-column widgets are composite widgets that allow the application to specify how child widgets of the composite should be laid out relative to each other and relative to the composite. The row-column widget positions its children in rows or columns. CwRowColumn widgets are frequently used to lay out groups of buttons. They can also be used to lay out widgets in a table.

Protocol

adjustLast: *aBoolean*
Extends the last row of children to the bottom edge of RowColumn (when orientation is horizontal) or extends the last column to the right edge of RowColumn (when orientation is vertical). This feature is disabled by setting XmNadjustLast to false.

buttonSet: *anInteger*
Specifies which button of a RadioBox or OptionMenu Pulldown submenu is initially set. The value is an integer n indicating the nth ToggleButton specified for a RadioBox or the nth PushButton specified for an OptionMenu Pulldown submenu. The first button specified is number 0.

entryAlignment: *anInteger*
Specifies the alignment type for CwLabel children.

Default: XmALIGNMENTBEGINNING (Left)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Left align children.
XmALIGNMENTCENTER (Center) - Center align children.
XmALIGNMENTEND (Right) - Right align children.

entryBorder: *anInteger*
Imposes a uniform border width upon all RowColumn's children.

Default: 0 (No Change)
Valid resource values:
0 (No Change) - Has no effect on widget borders.
1 (Uniform Borders) - Causes the widget to have a uniform border.

isAligned: *aBoolean*
Specifies text alignment for each item within the RowColumn widget; this only applies to items which are a subclass of CwLabel, and on some platforms, applies only to instances of CwLabel.

isHomogeneous: *aBoolean*
Indicates if the RowColumn widget should enforce exact homogeneity among the items it contains.

marginHeight: *anInteger*
Specifies the amount of blank space between the top edge of the RowColumn widget and the first item in each column, and the bottom edge of the RowColumn widget and the last item in each column.

marginWidth: *anInteger*
Specifies the amount of blank space between the left edge of the RowColumn widget and the first item in each row, and the right edge of the RowColumn widget and the last item in each row.

numColumns: *anInteger*
For vertically-oriented RowColumn widgets, this attribute indicates how many columns are built; the number of entries per column are adjusted to maintain this number of columns, if possible. For horizontally-oriented RowColumn widgets, this attribute indicates how many rows are built.

orientation: *anInteger*
Determines whether RowColumn layouts are row major or column major.

Default: XmVERTICAL (Column)
Valid resource values:
XmVERTICAL (Column) - In a column major layout, the children of the RowColumn are laid out in columns top to bottom within the widget.
XmHORIZONTAL (Row) - In a row major layout the children of the RowColumn are laid out in rows left to right within the widget.

packing: *anInteger*
Specifies how to pack the items contained within a RowColumn widget.

Default: XmPACKTIGHT (Pack Tight)
Valid resource values:
XmPACKTIGHT (Pack Tight) - Indicates that given the current major dimension, entries are placed one after the other until the RowColumn widget must wrap.
XmPACKCOLUMN (Pack Column) - Indicates that all entries are placed in identically sized boxes.
XmPACKNONE (Pack None) - Indicates that no packing is performed.

radioAlwaysOne: *aBoolean*
Forces the active ToggleButton to be automatically selected after having been unselected (if no other toggle was activated), if true. If false, the active toggle may be unselected. The default value is true.

radioBehavior: *aBoolean*
Specifies that the RowColumn widget should enforce a RadioBox-type behavior on all of its children which are ToggleButtons.

resizeHeight: *aBoolean*
Requests a new height if necessary, when set to true. When set to false, the widget does not request a new height regardless of any changes to the widget or its children.

resizeWidth: *aBoolean*
Requests a new width if necessary, when set to true. When set to false, the widget does not request a new width regardless of any changes to the widget or its children.

spacing: *anInteger*
Specifies the horizontal and vertical spacing between items contained within the RowColumn widget.

Callbacks & Events

Entry Callback
Supply a single callback routine for handling all items contained in a RowColumn widget. This disables the activation callbacks for all ToggleButton and PushButton widgets contained within the RowColumn widget.

Call data arguments:
widget - the value of widget. This is the widget that triggered the Entry Callback.
callbackData - the callData from the widget that triggered the Entry Callback.
data - the value of data.

Expose Callback
These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

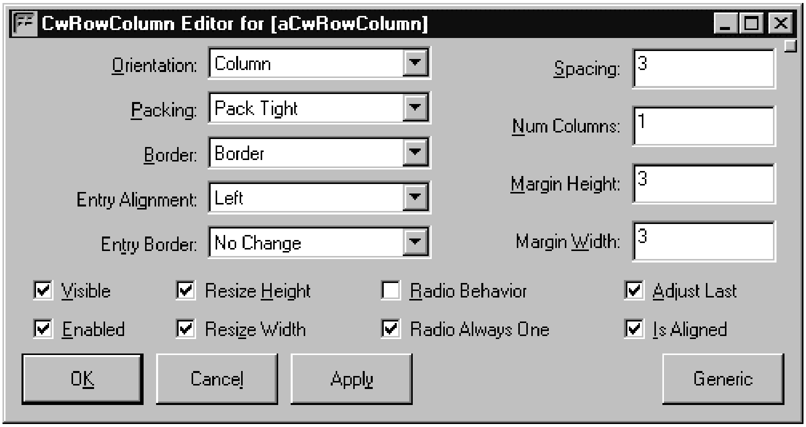
Map Callback
These callbacks are triggered when the window associated with the RowColumn widget is about to be mapped.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Simple Callback
These callbacks are triggered when a button is activated or when its value changes.

Unmap Callback
These callbacks are triggered after the window associated with the RowColumn widget has been unmapped.

Editor



CwRowColumn Editor for [aCwRowColumn]

Orientation: Column

Packing: Pack Tight

Border: Border

Entry Alignment: Left

Entry Border: No Change

Spacing: 3

Num Columns: 1

Margin Height: 3

Margin Width: 3

☑ Visible ☑ Resize Height □ Radio Behavior ☑ Adjust Last

☑ Enabled ☑ Resize Width ☑ Radio Always One ☑ Is Aligned

Adjust Last
Extends the last row of children to the bottom edge of RowColumn (when orientation is horizontal) or extends the last column to the right edge of RowColumn (when orientation is vertical). This feature is disabled by setting XmNadjustLast to false.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Button Set
Specifies which button of a RadioBox or OptionMenu Pulldown submenu is initially set. The value is an integer n indicating the nth ToggleButton specified for a RadioBox or the nth PushButton specified for an OptionMenu Pulldown submenu. The first button specified is number 0.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Entry Alignment
Specifies the alignment type for CwLabel children.

Center - Center align children.
Left - Left align children.
Right - Right align children.

Entry Border
Imposes a uniform border width upon all RowColumn's children.

No Change - Has no effect on widget borders.
Uniform Borders - Causes the widget to have a uniform border.

Is Aligned
Specifies text alignment for each item within the RowColumn widget; this only applies to items which are a subclass of CwLabel, and on some platforms, applies only to instances of CwLabel.

Is Homogeneous
Indicates if the RowColumn widget should enforce exact homogeneity among the items it contains.

Margin Height
Specifies the amount of blank space between the top edge of the RowColumn widget and the first item in each column, and the bottom edge of the RowColumn widget and the last item in each column.

Margin Width
Specifies the amount of blank space between the left edge of the RowColumn widget and the first item in each row, and the right edge of the RowColumn widget and the last item in each row.

Num Columns
For vertically-oriented RowColumn widgets, this attribute indicates how many columns are built; the number of entries per column are adjusted to maintain this number of columns, if possible. For horizontally-oriented RowColumn widgets, this attribute indicates how many rows are built.

Orientation
Determines whether RowColumn layouts are row major or column major.

Column - In a column major layout, the children of the RowColumn are laid out in columns top to bottom within the widget.
Row - In a row major layout the children of the RowColumn are laid out in rows left to right within the widget.

Packing
Specifies how to pack the items contained within a RowColumn widget.

Pack Column - Indicates that all entries are placed in identically sized boxes.
Pack None - Indicates that no packing is performed.
Pack Tight - Indicates that given the current major dimension, entries are placed one after the other until the RowColumn widget must wrap.

Radio Always One
Forces the active ToggleButton to be automatically selected after having been unselected (if no other toggle was activated), if true. If false, the active toggle may be unselected. The default value is true.

Radio Behavior
Specifies that the RowColumn widget should enforce a RadioBox-type behavior on all of its children which are ToggleButtons.

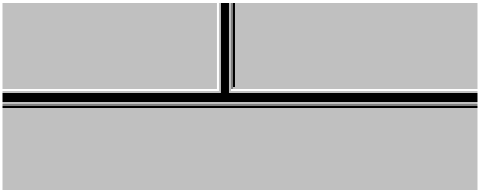
Resize Height
Requests a new height if necessary, when set to true. When set to false, the widget does not request a new height regardless of any changes to the widget or its children.

Resize Width
Requests a new width if necessary, when set to true. When set to false, the widget does not request a new width regardless of any changes to the widget or its children.

Spacing
Specifies the horizontal and vertical spacing between items contained within the RowColumn widget.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwSash



CwSash is a spilt bar widget providing a mechanism for resizing widgets within a composite. The split bar is represented by a thin bordered area, whose color may be specified by the user. The bar may be repositioned by pressing and holding mouse button one while the cursor is over the split bar, then moving the mouse to the desired position. Upon release of the mouse button, other widgets on the composite that are attached to it will update their sizes and positions.

Protocol

bottomLimitWidget: *aWidget*
For horizontally-oriented CwSash widgets, this attribute indicates the bottom most widget to which it is constrained. For vertically-oriented CwSash widgets, this attribute does not apply.

leftLimitWidget: *aWidget*
For vertically-oriented CwSash widgets, this attribute indicates the left most widget to which it is constrained. For horizontally-oriented CwSash widgets, this attribute does not apply.

orientation: *anInteger*
Displays Sash vertically or horizontally.

Default: XmVERTICAL (Vertical)
Valid resource values:
XmVERTICAL (Vertical) - Displays Scale vertically.
XmHORIZONTAL (Horizontal) - Displays Scale horizontally.

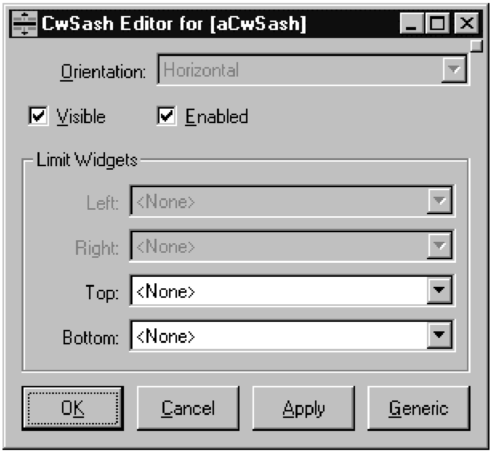
rightLimitWidget: *aWidget*
For vertically-oriented CwSash widgets, this attribute indicates the right most widget to which it is constrained. For horizontally-oriented CwSash widgets, this attribute does not apply.

topLimitWidget: *aWidget*
For horizontally-oriented CwSash widgets, this attribute indicates the top most widget to which it is constrained. For vertically-oriented CwSash widgets, this attribute does not apply.

Callbacks & Events

None

Editor



CwSash Editor for [aCwSash]

Orientation: Horizontal

☑ Visible ☑ Enabled

Limit Widgets

Left: <None>

Right: <None>

Top: <None>

Bottom: <None>

Bottom Limit Widget
Indicates the bottom most widget to which it is constrained.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Left Limit Widget
Indicates the left most widget to which it is constrained.

Orientation
Displays Scale vertically or horizontally. This attribute can only be set at creation time and will always be greyed out in the editor.

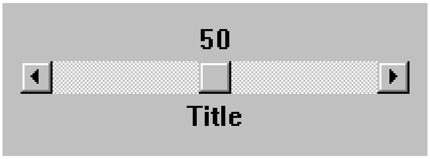
> Horizontal - Displays Scale horizontally.
> Vertical - Displays Scale vertically.

Right Limit Widget
Indicates the right most widget to which it is constrained.

Top Limit Widget
Indicates the top most widget to which it is constrained.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwScale



Scale widgets are specialized slider controls used to select a number from a range of allowed values. They may be either vertical or horizontal and can have their own arbitrary ranges.

Protocol

decimalPoints: *anInteger*
Specifies the number of decimal points to shift the slider value when displaying it.

getValue
Return the current slider position.

maximum: *anInteger*
Specifies the slider maximum value.

minimum: *anInteger*
Specifies the slider minimum value.

orientation: *anInteger*
Displays Scale vertically or horizontally.

Default: XmVERTICAL (Vertical)
Valid resource values:
XmVERTICAL (Vertical) - Displays Scale vertically.
XmHORIZONTAL (Horizontal) - Displays Scale horizontally.

processingDirection: *anInteger*
Specifies whether the value for maximum is on the right or left side of minimum for horizontal Scales or above or below minimum for vertical Scales.

Default: XmMAXONTOP (Top)
Valid resource values:
XmMAXONTOP (Top) - Maximum value is on top.
XmMAXONBOTTOM (Bottom) - Maximum value is on bottom.
XmMAXONLEFT (Left) - Maximum value is on left.
XmMAXONRIGHT (Right) - Maximum value is on right.

setValue: *anInteger*
Set the slider value to *anInteger*.

showValue: *aBoolean*
Specifies if a label for the current slider value should be displayed next to the slider. If it is true, the current slider value is displayed.

titleString: *aString*
Specifies the title text string to appear in the scale widget window.

value: *anInteger*
Specifies the slider current position along the scale, between minimum and maximum.

Callbacks & Events

Drag Callback

Specifies the list of callbacks that is called when the slider position changes as the slider is being dragged.

Call data arguments:
    value - the current value.

Expose Callback

These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback

These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback

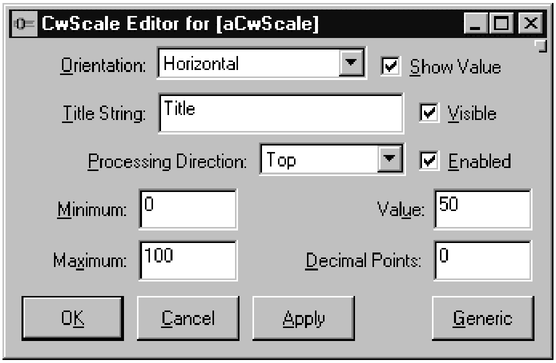
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback

These callbacks are triggered when the value of the slider has changed.

Call data arguments:
    value - the current value.

Editor



CwScale Editor for [aCwScale]

Orientation: Horizontal ☑ Show Value

Title String: Title ☑ Visible

Processing Direction: Top ☑ Enabled

Minimum: 0 Value: 50

Maximum: 100 Decimal Points: 0

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Decimal Points
Specifies the number of decimal points to shift the slider value when displaying it.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Maximum
Specifies the slider maximum value.

Minimum
Specifies the slider minimum value.

Orientation
Displays Scale vertically or horizontally.

Horizontal - Displays Scale horizontally.
Vertical - Displays Scale vertically.

Processing Direction
Specifies whether the value for maximum is on the right or left side of minimum for horizontal Scales or above or below minimum for vertical Scales.

Bottom - Maximum value is on bottom.
Left - Maximum value is on left.
Right - Maximum value is on right.
Top - Maximum value is on top.

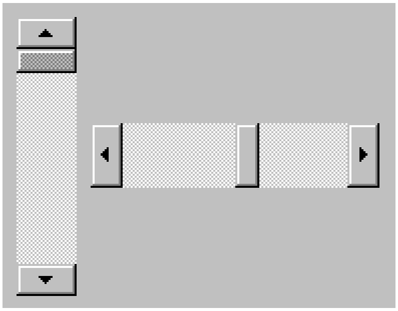
Show Value
Specifies if a label for the current slider value should be displayed next to the slider. If it is true, the current slider value is displayed.

Title String
Specifies the title text string to appear in the scale widget window.

Value
Specifies the slider current position along the scale, between minimum and maximum.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwScrollBar



Scrollbars are generic slider controls used to select a number from a range of allowed values. They may be either vertical or horizontal and can have their own arbitrary ranges.

Protocol

increment: *anInteger*
Specifies the amount to move the slider when the corresponding arrow is selected.

maximum: *anInteger*
Specifies the slider maximum value.

minimum: *anInteger*
Specifies the slider minimum value.

orientation: *anInteger*
Specifies whether the ScrollBar is displayed vertically or horizontally.

Default: XmVERTICAL (Vertical)
Valid resource values:
XmVERTICAL (Vertical) - Displays Scale vertically.
XmHORIZONTAL (Horizontal) - Displays Scale horizontally.

pageIncrement: *anInteger*
Specifies the amount to move the slider when selection occurs on the slide area.

processingDirection: *anInteger*
Specifies whether the value for maximum should be on the right or left side of minimum for horizontal ScrollBars or above or below minimum for vertical ScrollBars.

Default: XmMAXONBOTTOM (Bottom)
Valid resource values:

XmMAXONTOP (Top) - Maximum value is on top.
XmMAXONBOTTOM (Bottom) - Maximum value is on bottom.
XmMAXONLEFT (Left) - Maximum value is on left.
XmMAXONRIGHT (Right) - Maximum value is on right.

sliderSize: *anInteger*
Specifies the size of the slider between the values of 0 and maximum - minimum.

value: *anInteger*
Specifies the slider position between minimum and maximum.

Callbacks & Events

Decrement Callback

These callbacks are triggered when an arrow is selected which decreases the slider value by one increment.

Call data arguments:
value - the current value.

Drag Callback

These callbacks are triggered on each incremental change of position when the slider is being dragged.

Call data arguments:
value - the current value.

Increment Callback

These callbacks are triggered when an arrow is selected which increases the slider value by one increment.

Call data arguments:
value - the current value.

Page Decrement Callback

These callbacks are triggered when the slider area is selected and the slider value is decreased by one page increment.

Call data arguments:
value - the current value.

Page Increment Callback

These callbacks are triggered when the slider area is selected and the slider value is increased by one page increment.

Call data arguments:
value - the current value.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

To Bottom Callback

These callbacks are triggered when the user moves the slider to the bottom of the scroll bar.

Call data arguments:
    value - the current value.

To Top Callback

These callbacks are triggered when the user moves the slider to the top of the scroll bar.

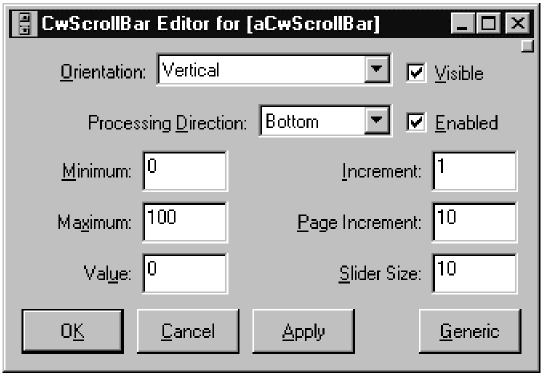
Call data arguments:
    value - the current value.

Value Changed Callback

These callbacks are triggered when the slider is released while being dragged.

Call data arguments:
    value - the current value.

Editor



CwScrollBar Editor for [aCwScrollBar]

Orientation: Vertical

☑ Visible

Processing Direction: Bottom

☑ Enabled

Minimum: 0

Increment: 1

Maximum: 100

Page Increment: 10

Value: 0

Slider Size: 10

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Increment

Specifies the amount to move the slider when the corresponding arrow is selected.

Maximum

Specifies the slider maximum value.

Minimum

Specifies the slider minimum value.

Orientation

Specifies whether the ScrollBar is displayed vertically or horizontally.

Horizontal - Displays Scale horizontally.
Vertical - Displays Scale vertically.

Page Increment

Specifies the amount to move the slider when selection occurs on the slide area.

Processing Direction

Specifies whether the value for maximum should be on the right or left side of minimum for horizontal ScrollBars or above or below minimum for vertical ScrollBars.

Bottom - Maximum value is on bottom.
Left - Maximum value is on left.
Right - Maximum value is on right.
Top - Maximum value is on top.

Slider Size

Specifies the size of the slider between the values of 0 and maximum - minimum.

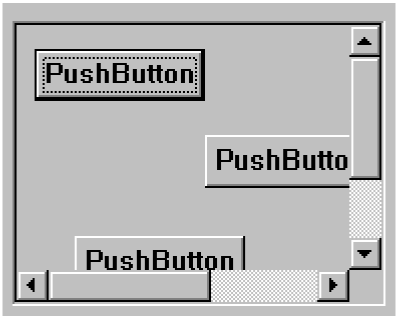
Value

Specifies the slider position between minimum and maximum.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwScrolledWindow



Scrolled windows are used to provide scrolling capabilities to other widgets. They may be used implicitly as with WbScrolledText or WbScrolledList or explicitly as with frames, row columns, icon and table widgets. For example, to make an EwTableList, EwIconList or similar widget scrollable, drop it into an exiting CwScrolledWindow instance. Note that without a CwScrolledWindow as a parent, these widgets will not have scrollbars. See Chapter 8 of the *IBM Smalltalk Programmer's Reference* for further details.

Protocol

clipBackgroundPixmap: *aCgPixmap*
Specifies a pixmap for tiling the background of the clip area. This will be used to fill areas within the scrolledWindow that are outside the workWindow. This is only used if the workWindow is smaller than the clip area.

scrollBarDisplayPolicy: *anInteger*
Controls the automatic placement of the ScrollBars. If it is set to As Needed and if scrollingPolicy is set to Automatic, ScrollBars will only be displayed if the workspace exceeds the clip area in one or both dimensions. A resource value of Static will cause the ScrolledWindow to display the ScrollBars whenever they are managed, regardless of the relationship between the clip area and the work area.

Default: XmSTATIC (Static)
Valid resource values:

XmSTATIC (Static) - Causes the ScrolledWindow to display the ScrollBars whenever they are managed, regardless of the relationship between the clip area and the work area.

XmASNEEDED (As Needed) - If scrollingPolicy is set to Automatic, ScrollBars will only be displayed if the workspace exceeds the clip area in one or both dimensions.

scrollingPolicy: *anInteger*
Performs automatic scrolling of the work area with no application interaction. If the value of this resource is Automatic, ScrolledWindow automatically creates the ScrollBars; sets the visual policy to Constant; and automatically moves the work area through the clip area in response to any user interaction with the ScrollBars. An application can also add its own callbacks to the ScrollBars. This allows the application to be notified of a scroll event without having to perform any layout procedures.

Default: XmAPPLICATIONDEFINED (Application Defined)
Valid resource values:

XmAUTOMATIC (Automatic) - ScrolledWindow automatically creates the ScrollBars; sets the visual policy to Constant; and automatically moves the work area through the clip area in response to any user interaction with the ScrollBars.

XmAPPLICATIONDEFINED (Application Defined) - The application is responsible for all aspects of scrolling. The ScrollBars must be created by the application, and it is responsible for performing any visual changes in the work area in response to user input.

visualPolicy: *anInteger*
Grows the ScrolledWindow to match the size of the work area, or it can be used as a static viewport onto a larger data space. If the visual policy is Variable, the ScrolledWindow will force the ScrollBar display policy to Static and allow the work area to grow or shrink at any time and will adjust its layout to accommodate the new size. When the policy is Constant, the work area will be allowed to grow or shrink as requested, but a clipping window will force the size of the visible portion to remain constant. The only time the viewing area can grow is in response to a resize from the ScrolledWindow’s parent.

Default: XmVARIABLE (Variable)
Valid resource values:

XmVARIABLE (Variable) - The ScrolledWindow will force the ScrollBar display policy to Static and allow the work area to grow or shrink at any time and will adjust its layout to accommodate the new size.

XmCONSTANT (Constant) - The work area will be allowed to grow or shrink as requested, but a clipping window will force the size of the visible portion to remain constant. The only time the viewing area can grow is in response to a resize from the ScrolledWindow’s parent.

Callbacks & Events

Expose Callback

These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback

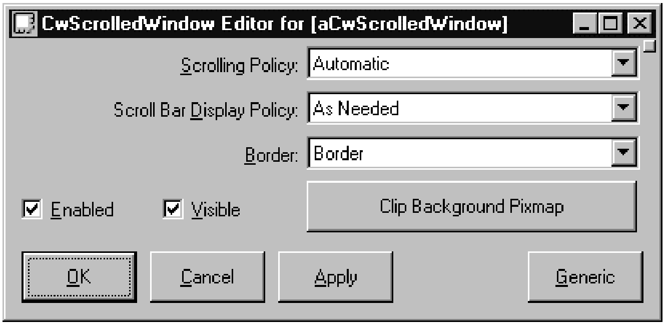
These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwScrolledWindow Editor for [aCwScrolledWindow]

Scrolling Policy: Automatic

Scroll Bar Display Policy: As Needed

Border: Border

☑ Enabled ☑ Visible

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Clip Background Pixmap
Specifies a pixmap for tiling the background of the clip area. This will be used to fill areas within the scrolledWindow that are outside the workWindow. This is only used if the workWindow is smaller than the clip area.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Scroll Bar Display Policy
Controls the automatic placement of the ScrollBars. If it is set to As Needed and if scrollingPolicy is set to Automatic, ScrollBars will only be displayed if the workspace exceeds the clip area in one or both dimensions. A resource value of Static will cause the ScrolledWindow to display the ScrollBars whenever they are managed, regardless of the relationship between the clip area and the work area.

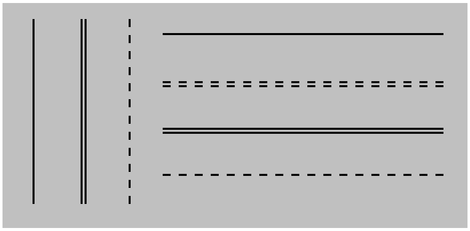
As Needed - If scrollingPolicy is set to Automatic, ScrollBars will only be displayed if the workspace exceeds the clip area in one or both dimensions.
Static - Causes the ScrolledWindow to display the ScrollBars whenever they are managed, regardless of the relationship between the clip area and the work area.

Scrolling Policy
Performs automatic scrolling of the work area with no application interaction. If the value of this resource is Automatic, ScrolledWindow automatically creates the ScrollBars; sets the visual policy to Constant; and automatically moves the work area through the clip area in response to any user interaction with the ScrollBars. An application can also add its own callbacks to the ScrollBars. This allows the application to be notified of a scroll event without having to perform any layout procedures.

Application Defined - The application is responsible for all aspects of scrolling. The ScrollBars must be created by the application, and it is responsible for performing any visual changes in the work area in response to user input.
Automatic - ScrolledWindow automatically creates the ScrollBars; sets the visual policy to Constant; and automatically moves the work area through the clip area in response to any user interaction with the ScrollBars.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwSeparator



Separators are used to visually separate groups of widgets from one another. The support a variety of visual styles and can be displayed horizontally or vertically.

Protocol

margin: *anInteger*
Specifies the space on the left and right sides between the border of the Separator and the line drawn for horizontal orientation. For vertical orientation, specifies the space on the top and bottom between the border of the Separator and the line drawn.

orientation: *anInteger*
Displays Separator vertically or horizontally.

Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - Displays Separator vertically.
XmHORIZONTAL (Horizontal) - Displays Separator horizontally.

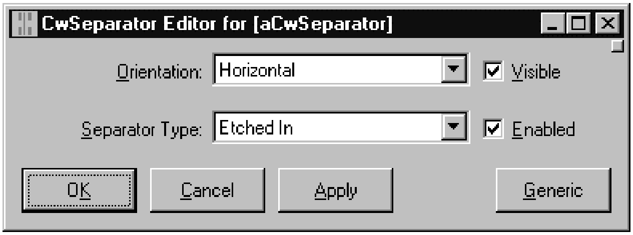
separatorType: *anInteger*
Specifies the type of line drawing to be done in the Separator widget.

Default: XmSHADOWETCHEDIN (Etched In)
Valid resource values:
XmNOLINE (No Line) - No line.
XmSINGLELINE (Single Line) - Draws Separator using a single line.
XmDOUBLELINE (Double Line) - Draws Separator using a double line.
XmSINGLEDASHEDLINE (Single Dashed Line) - Draws Separator using a single dashed line.
XmDOUBLEDASHEDLINE (Double Dashed Line) - Draws Separator using a double dashed line.
XmSHADOWETCHEDIN (Etched In) - Draws Separator using a double line giving the effect of a line etched into the window.

Callbacks & Events

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



CwSeparator Editor for [aCwSeparator]

Orientation: Horizontal ☑ Visible

Separator Type: Etched In ☑ Enabled

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Margin
Specifies the space on the left and right sides between the border of the Separator and the line drawn for horizontal orientation. For vertical orientation, specifies the space on the top and bottom between the border of the Separator and the line drawn.

Orientation
Displays Separator vertically or horizontally.

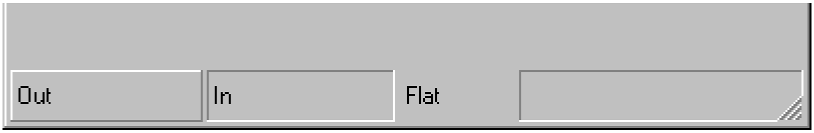
Horizontal - Displays Separator horizontally.
Vertical - Displays Separator vertically.

Separator Type
Specifies the type of line drawing to be done in the Separator widget.

Double Dashed Line - Draws Separator using a double dashed line.
Double Line - Draws Separator using a double line.
Etched In - Draws Separator using a double line giving the effect of a line etched into the window.
No Line - No line.
Single Dashed Line - Draws Separator using a single dashed line.
Single Line - Draws Separator using a single line.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwStatusBar



CwStatusBars provide an area, usually located along the bottom of a window, for an application to provide status information. A status bar can be divided up into a number of status panels. The appearance of status panels within status bars can be tailored by setting their shadow property. Using this property, you can have the entire status bar appear flat, “indented,” or “outdented,” similar to a push button. This widget is only available in the Windows version of VisualAge.

Protocol

createPanel: *theName* **argBlock:** *argBlock*
Create a CwPanel as a child of the receiver.

deleteAllItems
Delete all the items from the receiver and release all OS resources associated with the items.

deleteItem: *anItem*
Delete the item from the receiver and release all OS resources associated with the item.

items
Answer the items in the receiver.

labelString: *aString*
Specifies the label displayed when the receiver has showPanels false.

numItems
Answer the number of items in the receiver.

showPanels: *aBoolean*
If true, the statusbar shows all panels. If false, the status displays one large panel.

Callbacks & Events

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

CwPanel

Protocol

labelString: *aString*
Specifies the label that appears on the item.

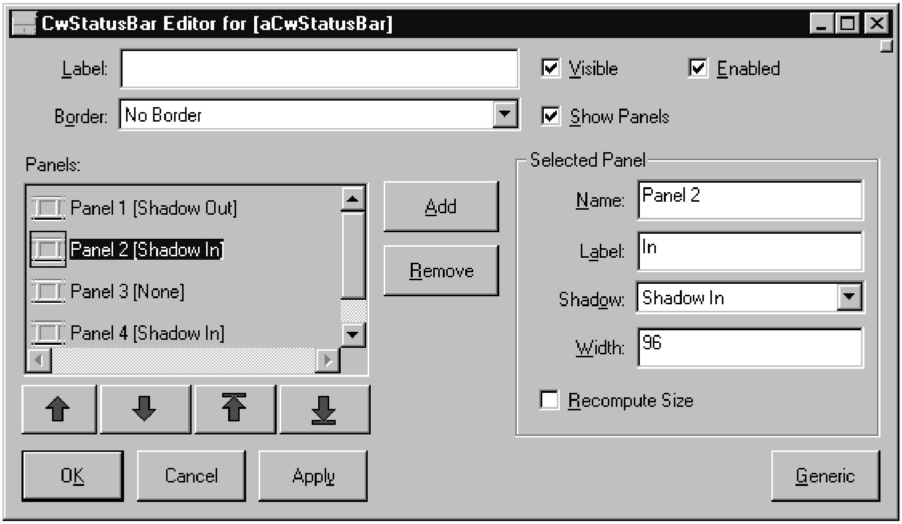
recomputeSize: *aBoolean*
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

shadowType: *anInteger*
Specifies the shadow type used by the panel.

> None - The panel displays no shadow.
>
> Shadow In - The panel appears to be inset into the status bar.
>
> Shadow Out - The panel appears to be raised above the status bar.

width: *anInteger*
Specifies the width of the panel.

Editor



CwStatusBar Editor for [aCwStatusBar]

Label:

☑ Visible ☑ Enabled

Border: No Border

☑ Show Panels

Panels:
- Panel 1 [Shadow Out]
- Panel 2 [Shadow In]
- Panel 3 [None]
- Panel 4 [Shadow In]

Selected Panel
- Name: Panel 2
- Label: In
- Shadow: Shadow In
- Width: 96
- ☐ Recompute Size

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

> Border - Causes the widget to have a border.
> No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Label String
Specifies the label displayed when the receiver has showPanels false.

Show Panels
If true, the statusbar shows all panels. If false, the status displays one large panel.

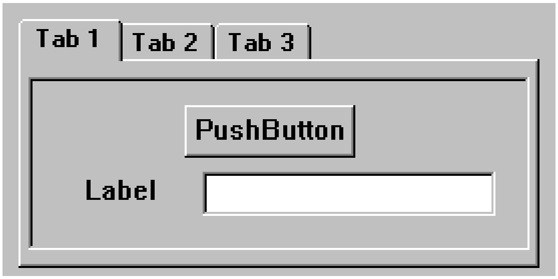
Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Panels
An array of CwPanels that are to be displayed as the status boxes.

Selected Panel
The currently edited CwPanel.

> **Label String**
> Specifies the label that appears on the panel.
>
> **Name**
> The name of the panel.
>
> **Recompute Size**
> Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.
>
> **Shadow Type**
> Specifies the shadow type used by the panel.
>
> > None - The panel displays no shadow.
> > Shadow In - The panel appears to be inset into the status bar.
> > Shadow Out - The panel appears to be raised above the status bar.
>
> **Width**
> Specifies the width of the panel.

CwTabStrip



CwTabStrip and CwTab are used to organize information better by defining multiple viewable “pages” for some area of a window. A CwTabStrip appears as a set of notebook page tabs. A user of your application navigates through the pages you have defined by clicking on their respective tabs. This causes the selected page to be brought to the front. CwTabs can be labeled with text, bitmaps, or both. This widget is only available in the Windows version of VisualAge.

Note that the CwTabStrip widget does *not* provide for any automatic linkage between a tab and a tab page. Tab pages must be set up as CwForms that are direct children of the CwTabStrip. They must then be managed and unmanaged programatically based upon tab selection (and example is provided in the WbTabStripExample class). For this reason, it is *strongly* recommended that you use the EwWINNotebook widget rather than the CwTabStrip widget.

Protocol

createTab: *theName* **argBlock:** *argBlock*
Create a CwTab as a child of the receiver.

deleteAllItems
Delete all the items from the receiver and release all OS resources associated with the items.

deleteItem: *anItem*
Delete the item from the receiver and release all OS resources associated with the item.

deleteItems: *items*
Delete the specified items from the receiver and release all OS resources associated with the items.

items
Answer the items in the receiver.

numItems
Answer the number of items in the receiver.

selectedItems
The selected tab items.

selectItem: *item* **notify:** *notify*
Select an item in the receiver and optionally invoke the selection callback.

showTips: *aBoolean*
Determines if the tabstrip will show tooltips on the tabs.

Callbacks & Events

Expose Callback
These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback
These callbacks are triggered when a tab is selected.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Single Selection Callback
These callbacks are triggered when a tab is selected.

CwTab

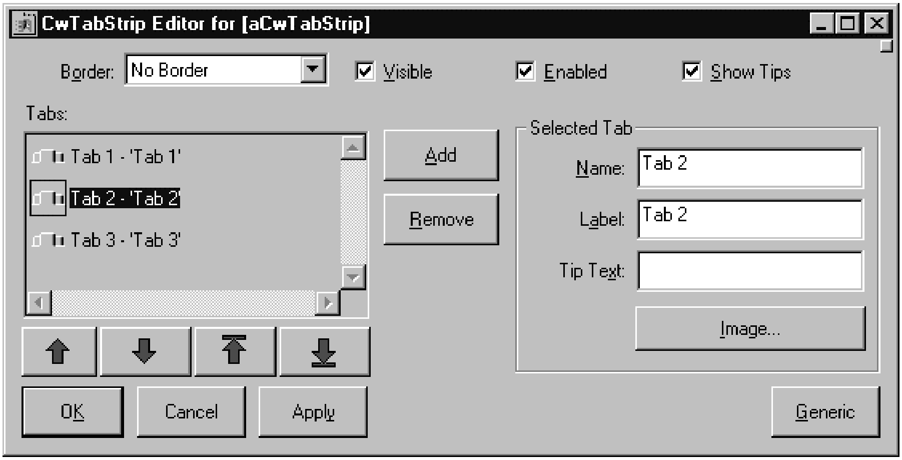
Protocol

image: *aPixmap*
Specifies the image displayed by the tab.

labelString: *aString*
Specifies the label that appears by th etab.

toolTipText: *aString*
Specifies text for the tab’s tooltip

Editor



CwTabStrip Editor for [aCwTabStrip]

Border: No Border

☑ Visible

☑ Enabled

☑ Show Tips

Tabs:

- Tab 1 - 'Tab 1'
- Tab 2 - 'Tab 2'
- Tab 3 - 'Tab 3'

Selected Tab

Name: Tab 2

Label: Tab 2

Tip Text:

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Show Tips
Determines if the tabstrip will show tooltips on the tabs.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Tabs
An array of CwTabs that are to be displayed as the status boxes.

Selected Tab
The currently edited CwTab.

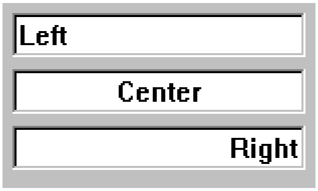
Name
The name of the panel.

Image
Specifies the image displayed by the tab.

Label
Specifies the label that appears by th etab.

Tip Text
Specifies text for the tab’s tooltip

CwText



Left

Center

Right

Text widgets provide text viewing and editing capabilities to the application. If the user types more text than can be accommodated within the field, it will automatically scroll.

Protocol

alignment: *anInteger*
Specifies the text alignment used by the widget.

Default: XmALIGNMENTBEGINNING (Left)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the left side of the line of text to be vertically aligned with the left edge of the widget window.
XmALIGNMENTCENTER (Center) - Causes the center of the line of text to be vertically aligned in the center of the widget window.
XmALIGNMENTEND (Right) - Causes the right side of the line of text to be vertically aligned with the right edge of the widget window.

clear
Clear the contents of the receiver.

clearSelection
Clear the selection.

columns: *anInteger*
Specifies the initial width of the text window measured in character spaces.

copySelection
Copy the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

cursorPosition: *anInteger*
Indicates the position in the text where the current insert cursor is to be located. Position is determined by the number of characters from the beginning of the text.

cutSelection
Cut the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

editable: *aBoolean*
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

editMode: *anInteger*
Specifies whether the widget supports single line or multi line editing of text.

Default: XmSINGLELINEEDIT (Single Line)
Valid resource values:
XmMULTILINEEDIT (Multi Line) - Multi line text edit.
XmSINGLELINEEDIT (Single Line) - Single line text edit.

getEditable
This message accesses the edit permission state of the Text widget.

getInsertionPosition
Return the position of the insert cursor. The return value is an integer number of characters from the beginning of the text buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getLastPosition
This message returns an Integer value that indicates the position of the last character in the text buffer. This is an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getSelection
Return a String containing the selection, or nil if there is no selection.

getSelectionPosition
Return a Point describing the selection position, where the x value is the start of the selection, and the y value is the end of the selection. The positions are an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getString
This message accesses the String value of the Text widget.

getTopCharacter
This message returns an Integer value that indicates the number of characters from the beginning of the text buffer. The first character position is 0.

insert: *position* **value:** *value*
Insert a String into the text. This message inserts a character string into the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This routine also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

insertAndShow: *position* **value:** *value*
Insert a String into the text and ensure that the text widget is scrolled such that the line containing the last new character inserted is visible. Vertical and/or horizontal scrolling may occur to accomplish this. This specification does not require that the text widget scroll horizontally but allows it. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This message also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

largeText: *aBoolean*
This is a hint that indicates that the receiver will be processing a large amount of text. If this flag is false, text operations may fail due to space limitations. If this hint is true, text operations will not fail.

lineDelimiter
Answer a String containing the line delimiting sequence used by the receiver. This value and number of characters may vary from platform to platform. The sequence is usually the standard end of line sequence for the platform. For example, on X/MOTIF this value is a String containing an ASCII LF character. On Windows, this value is a String containing ASCII CR and LF characters. All computations involving text positions operate

consistently with the number of characters in the lineDelimiter String. Thus an end of line takes up 1 character position on X and 2 character positions on Windows.

maxLength: *anInteger*
Specifies the maximum length of the text string that can be entered into text from the keyboard.

paste
Insert the clipboard selection into the text. Answer true if the operation is successful, or false if the text could not be retrieved from the clipboard.

readOnly
Set the readonly property of the receiver.

readWrite
Clear the readonly property of the receiver.

remove
Delete the selection.

replace: *fromPos* **toPos:** *toPos* **value:** *value*
Replace part of the receiver's text String. This message replaces part of the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. An example text replacement would be to replace the second and third characters in the text string. To accomplish this, the parameter fromPos must be 1 and toPos must be 3. To insert a string after the fourth character, both parameters, *fromPos* and *toPos*, must be 4. This message also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

rows: *anInteger*
Specifies the initial height of the text window measured in character heights.

scroll: *lines*
Scroll the text. This message scrolls text in a Text widget. *lines* specifies the number of lines of text to scroll. A positive value causes text to scroll upward; a negative value causes text to scroll downward.

scrollHorizontal: *aBoolean*
Adds a ScrollBar that allows the user to scroll horizontally through text.

scrollVertical: *aBoolean*
Adds a ScrollBar that allows the user to scroll vertically through text.

selectAll
Select the entire text of the receiver.

selectAtEnd
Place the gap selection at the end of the text.

selectedItem
Answer a String containing the text selected in clipboard format.

setEditable: *aBoolean*
This message sets the edit permission state of the Text widget. When set to True, the text string can be edited.

setHighlight: *positions* **mode:** *mode*
Set the text highlight. This message sets highlights text between the two specified character positions. The mode parameter determines the type of highlighting. Highlighting text merely changes the visual appearance of the text; it does not set the selection.

setInsertionPosition: *position*
Set the *position* of the insert cursor. This message sets the insertion cursor position of the Text widget.

setSelection: *positions*
Set the selection. This message sets the primary selection of the text in the widget. It also sets the insertion cursor position to the last position of the selection.

setString: *value*
This message sets the string value of the Text widget.

setTopCharacter: *topCharacter*
This message sets the position of the text at the top of the Text widget. If the editMode is XmMULTILINEEDIT, the line of text that contains *topCharacter* is displayed at the top of the widget without shifting the text left or right.

showPosition: *position*
Force text at the specified position to be displayed. This message forces text at the specified position to be displayed.

tabSpacing: *anInteger*
Indicates the tab stop spacing.

topCharacter: *anInteger*
Displays the position of text at the top of the window. Position is determined by the number of characters from the beginning of the text.

value: *aString*
Specifies the displayed text String.

verifyBell: *aBoolean*
Specifies whether the bell should sound when the verification returns without continuing the action.

wordWrap: *aBoolean*
Indicates that lines are to be broken at word breaks (i.e., the text does not go off the right edge of the window).

Callbacks & Events

Activate Callback
These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

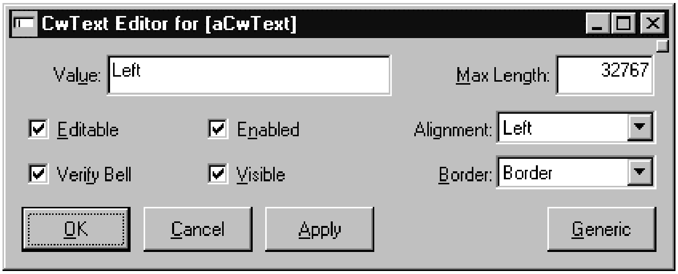
Modify Verify Callback
These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

Call data arguments:
text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback
These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



CwText Editor for [aCwText]

Value: Left

Max Length: 32767

☑ Editable ☑ Enabled Alignment: Left

☑ Verify Bell ☑ Visible Border: Border

Alignment

Specifies the text alignment used by the widget.

Left - Causes the left side of the line of text to be vertically aligned with the left edge of the widget window.

Center - Causes the center of the line of text to be vertically aligned in the center of the widget window.

Right - Causes the right side of the line of text to be vertically aligned with the right edge of the widget window.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.

No Border - Causes the widget to have no border.

Editable

Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Max Length

Specifies the maximum length of the text string that can be entered into text from the keyboard.

Value

Specifies the displayed text String.

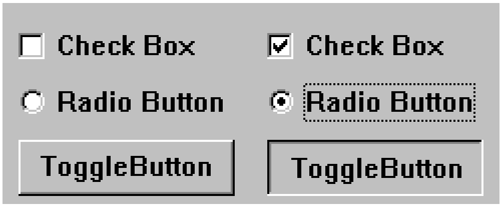
Verify Bell

Specifies whether the bell should sound when the verification returns without continuing the action.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwToggleButton



Toggle buttons have two states: on, and off. The state of a toggle button can be queried and changed using the `#getState` and `#setState:notify:` messages, respectively. Toggle buttons call their valueChanged callback when their state is changed. Toggle buttons are typically used to create radio button and check box groups. The toggle button indicatorType resource controls whether the toggle button has a radio button or a check box appearance.

Protocol

check
Set the receiver's state to checked (true).

click
Programatically click the button.

getState
Answers the state of the receiver.

indicatorOn: *aBoolean*
Specifies that a toggle indicator is drawn to the left of the toggle text or pixmap when set to true. When set to false, no space is allocated for the indicator, and it is not displayed.

indicatorType: *anInteger*
Specifies if the indicator is a 1-of or N-of indicator.

Default: XmNOFMANY (N-of-Many)
Valid resource values:
XmNOFMANY (N-of-Many) - Causes the ToggleButton to look like a CheckBox
XmONEOFMANY (1-of-Many) - Causes the ToggleButton to look like a RadioButton

labelInsensitivePixmap: *aCgPixmap*
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

labelPixmap: *aCgPixmap*
Specifies the pixmap when label type is Pixmap.

labelString: *aString*
Specifies the label string when the label type is String.

labelType: *anInteger*
Specifies the label type.

Default: XmSTRING (Text)
Valid resource values:
XmPIXMAP (Pixmap) - Causes the label to display a pixmap
XmSTRING (Text) - Causes the label to display text

mnemonic: *aCharacter*
Provides the user with alternate means for selecting a button.

recomputeSize: *aBoolean*
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

set: *aBoolean*
Displays the button in its selected state if set to true.

setState: *state* **notify:** *notify*
Sets or changes the CwToggleButton's current state. *state* specifies a Boolean value that indicates whether the ToggleButton state is selected or unselected. If true, the button state is selected; if false, the button state is unselected. *notify* indicates whether XmNvalueChangedCallback is called; it can be either true or false.

turnOff
Set the receiver's state to checked (false).

turnOn
Set the receiver's state to checked (true).

uncheck
Set the receiver's state to checked (false).

Callbacks & Events

Arm Callback

These callbacks are triggered when the button is armed. Buttons are armed and appear pressed whenever the moused is pressed within the button and not yet released. If the mouse is moved outside of the button while still pressed, the button will be disarmed and appear unpressed. If the mouse is released while still in the button, the button is activated

Disarm Callback
These callbacks are triggered when the button is disarmed. Buttons are disarmed whenever the mouse is moved outside of the button after it has been armed. Moving the mouse back over the button while the mouse button is still down will cause the button to become rearmed.

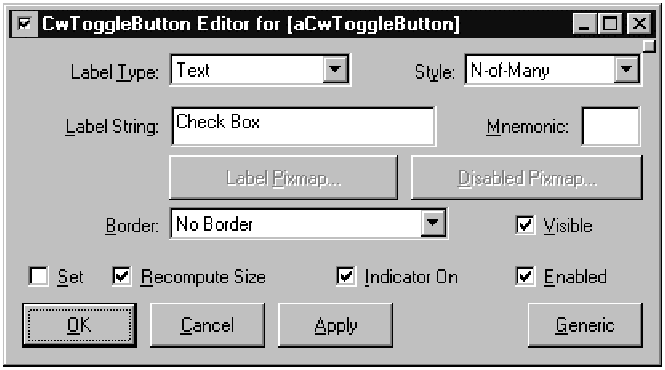
Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback
These callbacks are triggered when the ToggleButton value is changed.

Call data arguments:

set - a Boolean value indicating if the CwToggleButton is toggle on (true) or off (false).

Editor



CwToggleButton Editor for [aCwToggleButton]

Label Type: Text
Style: N-of-Many
Label String: Check Box
Mnemonic:
Border: No Border
☑ Visible
□ Set
☑ Recompute Size
☑ Indicator On
☑ Enabled

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Indicator On
Specifies that a toggle indicator is drawn to the left of the toggle text or pixmap when set to true. When set to false, no space is allocated for the indicator, and it is not displayed.

Indicator Type
Specifies if the indicator is a 1-of or N-of indicator.

1-of-Many - Causes the ToggleButton to look like a RadioButton
N-of-Many - Causes the ToggleButton to look like a CheckBox

Label Insensitive Pixmap
Specifies a pixmap used as the button face if label type is Pixmap and the button is insensitive.

Label Pixmap
Specifies the pixmap when label type is Pixmap.

Label String
Specifies the label string when the label type is String.

Label Type
Specifies the label type.

Icon - Causes the label to display an icon
Pixmap - Causes the label to display a pixmap
Text - Causes the label to display text

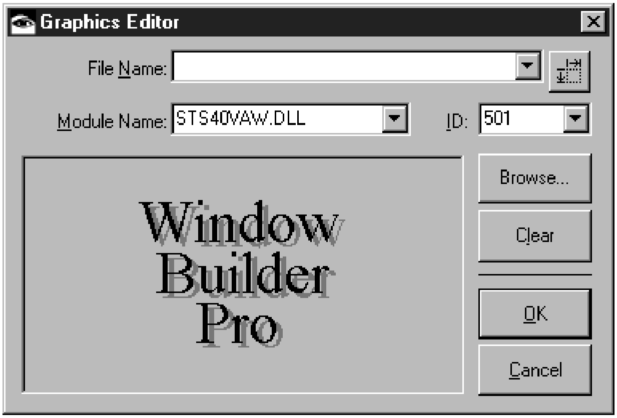
Mnemonic
Provides the user with alternate means for selecting a button.

Recompute Size
Specifies a Boolean value that indicates whether or not the widget always attempts to be big enough to contain the label.

Set
Displays the button in its selected state if set to true.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Graphics Editor


Graphics Editor

File Name:

Module Name: STS40VAW.DLL

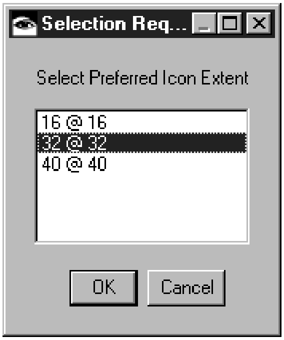
ID: 501

File Name

Specifies the file name of the graphic file to be used as the label for the widget. A variety of graphics file formats are supported including standard bitmaps (BMP files), icons (ICO files), PCX file and TIFF files. If the file name does not include a path, the file is assumed to reside in the local directory or in the system bitmaps directory.

Preferred Icon Extent

Specifies preferred icon extent which is used in the event that the icon file contains multiple icon resources of different sizes.


Selection Req...

Select Preferred Icon Extent

16 @ 16
32 @ 32
40 @ 40

Module Name

Specifies the name of the module (DLL) containing the bitmap or icon to be used as the label for the widget

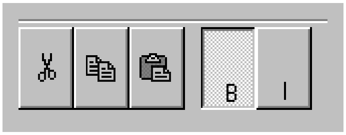
ID

Specifies the ID of the of the bitmap in the specified module.

Browse
Opens a file dialog from which a graphics file may be selected.

Clear
Clears any graphic choice that has been made.

CwToolBar



CwToolBar is a collection of buttons, CwToolButtons, which is typically used to provide quick access to an application’s frequently used commands and functions. CwToolButtons can be labeled with text, images, or both. In addition to CwToolButtons, there are separators which provide extra space between CwToolButtons and allow for logical grouping. CwToolButtons have slightly different behavior from CwPushButtons in that they cannot be resized and that they will always orient themselves to the upper-left position in the CwToolBar. Separators can be resized to define extra space between CwToolButtons. This widget is only available in the Windows version of VisualAge.

Protocol

createToolButton: *theName* **argBlock:** *argBlock*
Create a CwToolButton as a child of the receiver.

deleteAllItems
Delete all the items from the receiver and release all OS resources associated with the items.

deleteItem: *anItem*
Delete the item from the receiver and release all OS resources associated with the item.

deleteItems: *items*
Delete the specified items from the receiver and release all OS resources associated with the items.

items
Answer the items in the receiver.

numItems
Answer the number of items in the receiver.

showTips: *aBoolean*
Determines if the tabstrip will show tooltips on the tabs.

Callbacks & Events

Expose Callback
These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback
These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback
These callbacks are triggered when a tab is selected.

Call data arguments:
doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Single Selection Callback
These callbacks are triggered when a tab is selected.

CwToolButton

Protocol

buttonType: *anInteger*
Specifies the appearance and behavior of the button.

Default: XmPUSHBUTTON (Push Button)
Valid resource values:
XmPUSHBUTTON (Push Button) - The button is a push button.
XmCHECKBUTTON (Check Button) - The button is a check button.
XmRADIOBUTTON (Radio Button) - The button remains pressed until another button in the group is pressed. Exactly one button in the group can be pressed at any one moment. A radio group is defined as consecutive buttons with the XmRADIOBUTTON type.
XmSEPARATOR (Separator) - The button functions as a separator.

image: *aPixmap*
Specifies the image displayed by the button.

labelString: *aString*
Specifies the label that appears on the item.

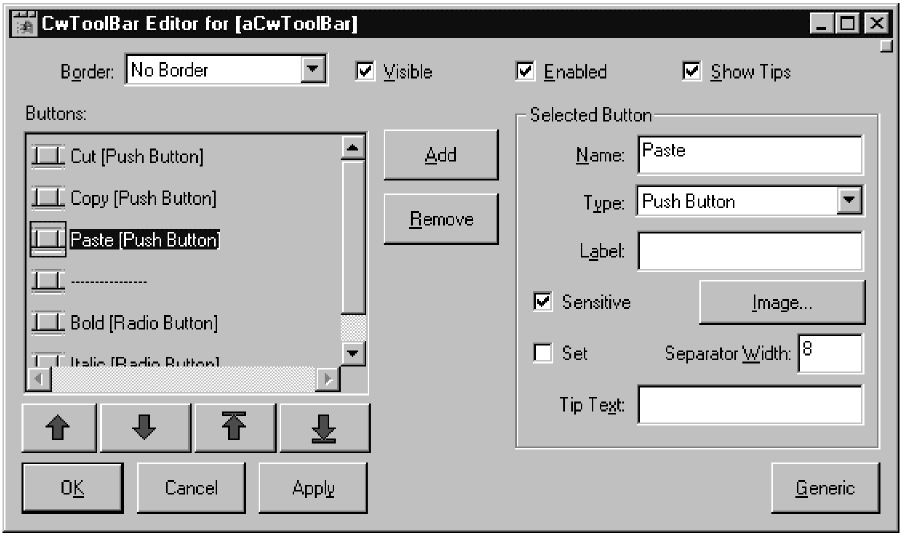
sensitive: *aBoolean*
Specifies whether a button will react to input events.

separatorWidth: *anInteger*
Specifies width of the separator when the style is XmSEPARATOR.

set: *aBoolean*
Displays the button in its selected state if set to true.

toolTipText: *aString*
Specifies text for the button’s tooltip

Editor



CwToolBar Editor for [aCwToolBar]

Border: No Border
☑ Visible
☑ Enabled
☑ Show Tips

Buttons:
- Cut [Push Button]
- Copy [Push Button]
- Paste [Push Button]
- ---------------
- Bold [Radio Button]
- Italic [Radio Button]

Selected Button
- Name: Paste
- Type: Push Button
- Label:
- ☑ Sensitive
- ☐ Set
- Separator Width: 8
- Tip Text:

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Show Tips
Determines if the tabstrip will show tooltips on the tabs.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Buttons
An array of CwToolButtons that are to be displayed within the toolbar.

Selected Button
The currently edited CwToolButton.

Button Type
Specifies the appearance and behavior of the button.

Push Button - The button is a push button.
Check Button - The button is a check button.
Radio Button - The button remains pressed until another button in the group is pressed. Exactly one button in the group can be pressed at any one moment. A radio group is defined as consecutive buttons with the XmRADIOBUTTON type.
Separator - The button functions as a separator.

Name
The name of the button.

Image
Specifies the image displayed by the button.

Label
Specifies the label displayed by the button.

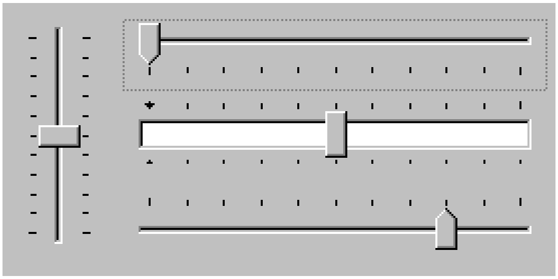
Sensitive
Specifies whether a button will react to input events.

Separator Width
Specifies width of the separator when the button type is Separator

Set
Displays the button in its selected state if set to true.

Tip Text
Specifies text for the button’s tooltip

CwTrackBar



CwTrackBar is a visual control with a sliding marker and optional tick marks. It can be oriented either horizontally or vertically. It is useful when you wish the user of your application to select a discrete value from a range of values. The ticks marks can appear above or below the sliding marker for horizontally oriented track bars, to the left or right of the sliding marker for vertically oriented track bars, or can be omitted all together. You specify the minimum and maximum values for the track bar which define the range of values your user can select, and an increment, that is the granularity of movement of the marker. You can also specify the spacing of the tick marks. For example, a track bar with a minimim of 0, maximum of 10, and increment of 1 will permit the user to select integers between 0 and 10 inclusive. The same track bar with an increment of 2 will permit selection only of even integers. You obtain and set the value of the track bar through its value attribute. This widget is only available in the Windows version of VisualAge.

Protocol

increment: *anInteger*
Specifies the number of ticks the slider will move when the left or right arrows are pressed.

maximum: *anInteger*
Specifies the maximum value of the slider.

minimum: *anInteger*
Specifies the minimum value of the slider.

orientation: *anInteger*
Determines whether the slider is oriented horizontally or vertically.

Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - The slider moves vertically.
XmHORIZONTAL (Horizontal) - The slider moves horizontally.

pageIncrement: *anInteger*
Specifies the number of ticks the slider will move when the PAGEUP or PAGEDOWN keys are pressed, or when the mouse is clicked to the left or right of the slider.

selection: *aPoint*
Specifies the start and end of the select range in the slider. This is valid if the select range is enabled.

showSelection: *aBoolean*
Determine if the slider can have a select range.

showTickBottom: *aBoolean*
Specifies the positioning of the tick marks displayed on the receiver. If true, tick marks are positioned along the bottom of the slider if it is horizontal, or along the right side if it is vertical. If false, no tick marks are placed on the bottom/right.

showTickTop: *aBoolean*
Specifies the positioning of the tick marks displayed on the receiver. If true, tick marks are positioned along the top of the slider if it is horizontal, or along the left side if it is vertical. If false, no tick marks are placed on the top/left.

tickFrequency: *anInteger*
Specifies the frequency of the tick marks on a slider in relation to its range.

value: *anInteger*
Specifies the current position of the slider.

Callbacks & Events

Drag Callback

These callbacks are triggered when the slider is moved either by clicking or using the keyboard.

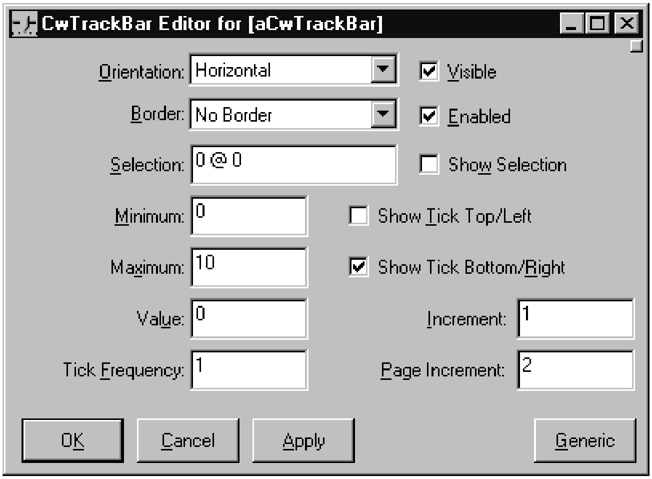
Call data arguments:
value - the current value.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback
These callbacks are triggered when the slider is released while being dragged.

Call data arguments:
value - the current value.

Editor



CwTrackBar Editor for [aCwTrackBar]

Orientation: Horizontal
Border: No Border
Selection: 0 @ 0
Minimum: 0
Maximum: 10
Value: 0
Tick Frequency: 1

☑ Visible
☑ Enabled
□ Show Selection
□ Show Tick Top/Left
☑ Show Tick Bottom/Right

Increment: 1
Page Increment: 2

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Increment
Specifies the number of ticks the slider will move when the left or right arrows are pressed.

Maximum
Specifies the maximum value of the slider.

Minimum
Specifies the minimum value of the slider.

Orientation
Determines whether the slider is oriented horizontally or vertically.

> Vertical - The slider moves vertically.
> Horizontal - The slider moves horizontally.

Page Increment
Specifies the number of ticks the slider will move when the PAGEUP or PAGEDOWN keys are pressed, or when the mouse is clicked to the left or right of the slider.

Selection
Specifies the start and end of the select range in the slider. This is valid if the select range is enabled.

Show Selection
Determine if the slider can have a select range.

Show Tick Bottom/Right
Specifies the positioning of the tick marks displayed on the receiver. If true, tick marks are positioned along the bottom of the slider if it is horizontal, or along the right side if it is vertical. If false, no tick marks are placed on the bottom/right.

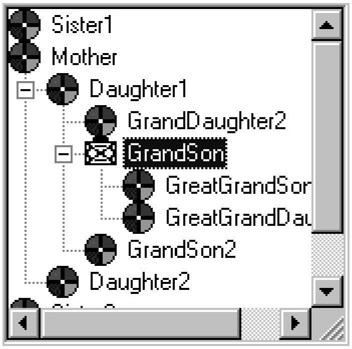
Show Tick Top/Left
Specifies the positioning of the tick marks displayed on the receiver. If true, tick marks are positioned along the top of the slider if it is horizontal, or along the left side if it is vertical. If false, no tick marks are placed on the top/left.

Tick Frequency
Specifies the frequency of the tick marks on a slider in relation to its range.

Value
Specifies the current position of the slider.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

CwTreeView



CwTreeView is a visual control useful for displaying information in a hierarchical organization. It consists of a list of objects. These objects can be comprised of other objects, which in turn can be comprised of other objects, and so on. A user navigates through the lists by expanding and collapsing objects to view and to hide their component objects. There are a number of attributes you can set to control the behavior of a tree view. You can define the default bitmaps for selected and non-selected objects. You can set the level of indentation for the display of an object's component objects. You can specify to have lines from an object to its component objects or lines to the root object shown or not shown.. This widget is only available in the Windows version of VisualAge.

Protocol

indentation: *anInteger*
Specifies the amount each node is indented.

showImages: *aBoolean*
Determines if the receiver displays images. Labels are always displayed.

showLines: *aBoolean*
Determines whether to display lines between the nodes.

showPlusMinus: *aBoolean*
Determines if the receiver displays plus/minus icons. Labels are always displayed.

showRootLines: *aBoolean*
Determines whether to display lines between the roots.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Collapse Callback

These callbacks are triggered when a node is collapsed.

Call data arguments:
    value - the current value.

Expand Callback

These callbacks are triggered when a node is expanded.

Call data arguments:
    value - the current value.

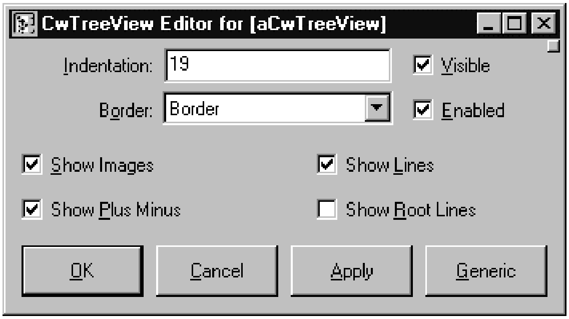
Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Single Selection Callback

These callbacks are triggered when an item is selected.

Editor



CwTreeView Editor for [aCwTreeView]

Indentation: 19
Border: Border
☑ Visible
☑ Enabled
☑ Show Images
☑ Show Lines
☑ Show Plus Minus
☐ Show Root Lines

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Indentation
Specifies the amount each node is indented.

Show Images
Determines if the receiver displays images. Labels are always displayed.

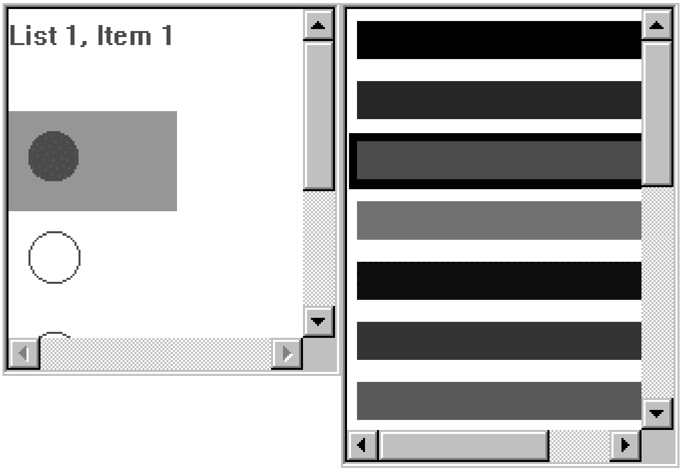
Show Lines
Determines whether to display lines between the nodes.

Show Plus Minus
Determines if the receiver displays plus/minus icons. Labels are always displayed.

Show Root Lines
Determines whether to display lines between the roots.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwDrawnList



The application-drawn list widget (EwDrawnList) allows an application to draw arbitrary graphics to represent each object in the list. It combines list behavior with a drawing area-based widget..

An application hooks the displayCallback to draw the items in the list. If the items in the list have different sizes, an application should hook the measureCallback to specify the height of each individual item in the list. If all items have the same height, the itemHeight resource can be used to specify the height in pixels.

The applicationDrawnStates resource allows for the specification of visuals for any of the emphasis states in the list, such as selection emphasis or cursored emphasis. Applications can choose to allow the drawn list to provide these emphasis visuals. In the following code, a list of CgFontStructs is added to a drawn list, and each font name in the list is drawn with the font that it describes.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnStates: *anInteger*
Describes which visual states will be custom drawn by the application in the Display Callback.

Default: XmDRAWNONE (No Custom Drawing)
Valid resource values:
XmDRAWNONE (No Custom Drawing) - No Custom Drawing.
XmDRAWSELECTION (Draw Selection) - Custom draw the selection.
XmDRAWCURSORED (Draw Cursored) - Custom draw the cursored item.
XmDRAWSELECTION | XmDRAWCURSORED (Draw Selection+Cursored) - Custom draw the selection and cursored item.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

itemWidth: *anInteger*
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

Callbacks & Events

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Display Callback

These callbacks are triggered when an item is to be drawn.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Measure Callback

These callbacks are triggered in order to obtain the height of an item. The application specifies the height by setting the height field of the callData to the Integer pixel height of the item.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may 'undo' the selection change by setting the doit field of the callData to false.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

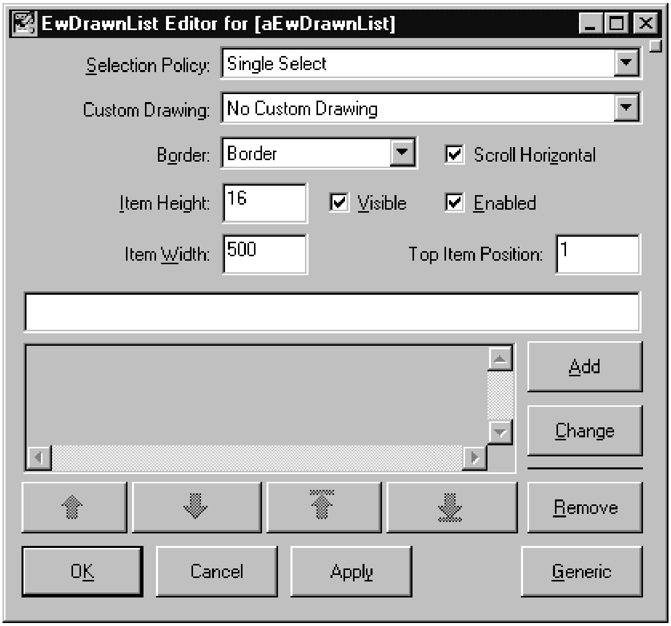
Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Editor



EwDrawnList Editor for [aEwDrawnList]

Selection Policy: Single Select

Custom Drawing: No Custom Drawing

Border: Border ☑ Scroll Horizontal

Item Height: 16 ☑ Visible ☑ Enabled

Item Width: 500 Top Item Position: 1

Application Drawn States (Custom Drawing)
Describes which visual states will be custom drawn by the application in the Display Callback.

Draw Cursored - Custom draw the cursored item.
Draw Selection - Custom draw the selection.
Draw Selection+Cursored - Custom draw the selection and cursored item.
No Custom Drawing - No Custom Drawing.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Item Height
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Item Width
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

Items
An array of objects that are to be displayed as the list items.

Scroll Horizontal
This resource specifies whether a horizontal scroll bar should be used for the list.

Selection Policy
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

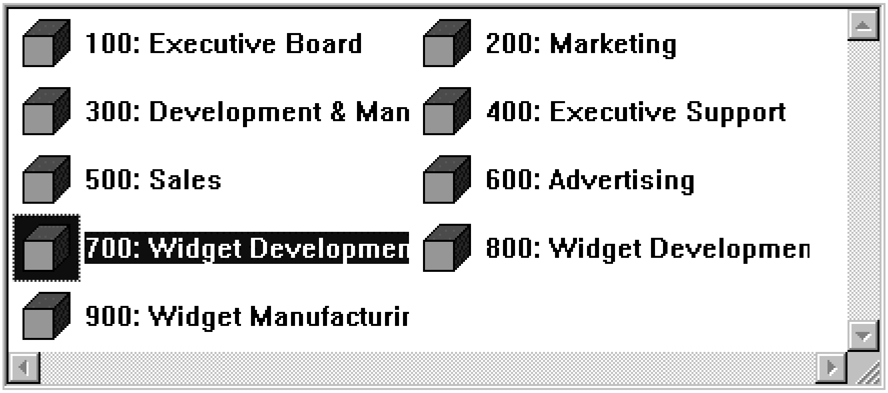
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Read Only Select - Allows navigation, but no selection or callbacks.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwFlowedIconList



This class provides a vertical, multi-column list of items with an icon and a label for each item. The items in the list are typically actual application objects, not strings or arrays.

The items are arranged in rows with as many items per row as can fit in the available width. Each items width and height are determined by the XmNitemWidth and XmNitemHeight resources. Thus, the dimensions of the rows and columns are fixed.

The widget then fires the visualInfo callback to obtain the label and icon for each item in the list as well as its inUse state. It only fires this callback for items which are visible on the screen. The application *must* hook the visualInfo callback and set the callData label to be the string (or other renderable object) which is to be the label for the item in callData item. The application *must* set the callData icon to be the CgIcon (or other renderable object) which is to be the icon for the item in callData item.

If the application desires a view with only a single icon or label per item, this can be achieved by setting the icon to nil for each item, setting the label to the desired visual for each item, setting the labelOrientation to XmRIGHT.

The objects used for item labels are typically Strings, and those used for item icons are typically CgIcons, but this is not a requirement. It is possible for an application to use any object as the label and icon for an item. Any item which understands `#ewHeightUsing:`, `#ewWidthUsing:` and `#ewDrawUsing:` (called a "renderable object") can be used for the label and icon. Since String and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

For example, if an application wishes to have different item's labels be different colors, it should implement a "ColoredString" class. This could then override the default rendering protocol and always render the string in a particular color, overriding the color in the gc of the EwRenderContext given. The same approach can be applied to handle item labels with different fonts, fixed alignments, etc.

Label editing can take place under program control only. When a label is about to be edited, the widget fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData `doit: true` to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the item label about to be edited. For example, if the item label contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the widget fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The label is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

emphasisPolicy: *anInteger*
Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Default: XmSEPARATE (Separate)
Valid resource values:
  XmTOGETHER (Together) - Draw the icon and label emphasis as one single rectangle.
  XmSEPARATE (Separate) - Draw the icon and label emphasis as two separate rectangles.

innerMargin: *anInteger*
Specifies the margin width to be used between each item's icon and its label.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

itemWidth: *anInteger*
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

labelOrientation: *anInteger*
Specifies the label position associated with the items.

Default: XmRIGHT (Right)
Valid resource values:
XmRIGHT (Right) - Position the label to the right of the icon.
XmBOTTOM (Bottom) - Position the label beneath the icon.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

Callbacks & Events

Begin Edit Callback

These callbacks are triggered when an item is about to be edited.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback

These callbacks are triggered when an item’s background needs to be drawn.

Call data arguments:
item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.
value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.

renderContext - the render context to be used in drawing the item’s background
region - the region of the item whose background needs to be drawn.

End Edit Callback
These callbacks are triggered when an item is done being edited.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Extended Selection Callback
These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Modify Verify Callback
These callbacks are triggered when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

Call data arguments:
doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback
These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Single Selection Callback
These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

- item - the item which is the selected item.
- itemPosition - the integer position of the selected item in the list.
- selectedItemCount - the integer number of selected items.
- selectedItemPositions - a Collection containing the integer positions of the selected items.
- selectedItems - a Collection of items which are the selected items.

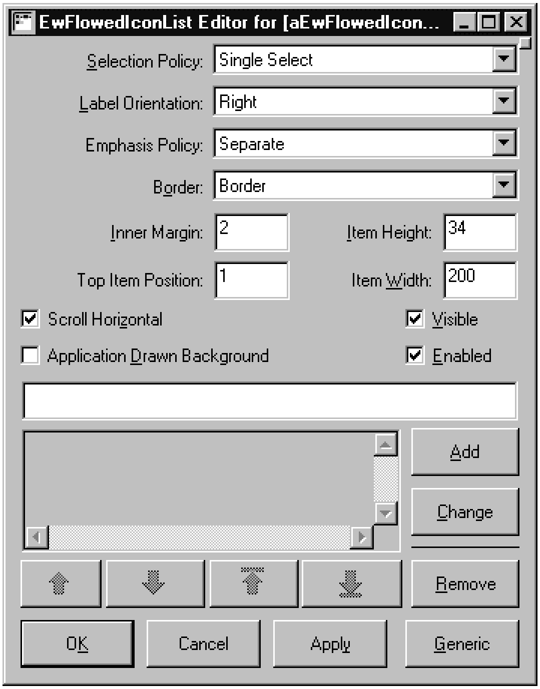
Visual Info Callback

These callbacks are triggered when an item's icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:

- item - the item which is the selected item.
- itemPosition - the integer position of the selected item in the list.
- isInUse - the inUse status (Boolean) which is the default to be used for the item in the callback.
- icon - the icon which is the default icon to be used for the item in the callback.
- label - the label which is the default label to be used for the item in the callback.

Editor



EwFlowedIconList Editor for [aEwFlowedIcon...

Selection Policy: Single Select

Label Orientation: Right

Emphasis Policy: Separate

Border: Border

Inner Margin: 2

Item Height: 34

Top Item Position: 1

Item Width: 200

☑ Scroll Horizontal

☑ Visible

☐ Application Drawn Background

☑ Enabled

Application Drawn Background
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Emphasis Policy
Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Separate - Draw the icon and label emphasis as two separate rectangles.
Together - Draw the icon and label emphasis as one single rectangle.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Inner Margin
Specifies the margin width to be used between each item's icon and its label.

Item Height
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Item Width
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

Items
An array of objects that are to be displayed as the list items.

Label Orientation
Specifies the label position associated with the items.

Bottom - Position the label beneath the icon.
Right - Position the label to the right of the icon.

Scroll Horizontal
This resource specifies whether a horizontal scroll bar should be used for the list.

Selection Policy

Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Read Only Select - Allows navigation, but no selection or callbacks.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

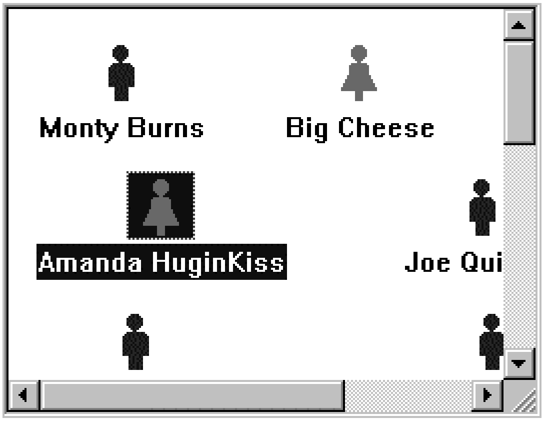
Top Item Position

Specifies the Integer position of the item that is the first visible item in the list.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwIconArea



This class provides a free-form display of items with an icon and a label for each item. The items in the list are typically actual application objects, not strings or arrays.

The widget then fires the visualInfo callback to obtain the label and icon for each item in the list as well as its inUse state. It only fires this callback for items which are visible on the screen. The application *must* hook the visualInfo callback and set the callData label to be the string (or other renderable object) which is to be the label for the item in callData item. The application *must* set the callData icon to be the CgIcon (or other renderable object) which is to be the icon for the item in callData item.

If the application desires a view with only a single icon or label per item, this can be achieved by setting the icon to nil for each item, setting the label to the desired visual for each item, setting the labelOrientation to XmRIGHT.

The objects used for item labels are typically Strings, and those used for item icons are typically CgIcons, but this is not a requirement. It is possible for an application to use any object as the label and icon for an item. Any item which understands `#ewHeightUsing`:, `#ewWidthUsing`: and `#ewDrawUsing`: (called a "renderable object") can be used for the label and icon. Since String and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

For example, if an application wishes to have different item's labels be different colors, it should implement a "ColoredString" class. This could then override the default rendering protocol and always render the string in a particular color, overriding the color in the gc of the EwRenderContext given. The same approach can be applied to handle item labels with different fonts, fixed alignments, etc.

Label editing can take place under program control only. When a label is about to be edited, the widget fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData `doit: true` to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the item label about to be edited. For example, if the item label contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the widget fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The label is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

innerMargin: *anInteger*
Specifies the margin width to be used between each item's icon and its label.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

labelOrientation: *anInteger*
Specifies the label position associated with the items.

Default: XmRIGHT (Right)
Valid resource values:
XmRIGHT (Right) - Position the label to the right of the icon.
XmBOTTOM (Bottom) - Position the label beneath the icon.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.
XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

Callbacks & Events

Begin Edit Callback
These callbacks are triggered when an item is about to be edited.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback

These callbacks are triggered when an item's background needs to be drawn.

Call data arguments:

item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.
value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.
renderContext - the render context to be used in drawing the item's background
region - the region of the item whose background needs to be drawn.

End Edit Callback

These callbacks are triggered when an item is done being edited.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:
  item - the item which is the selected item.
  itemPosition - the integer position of the selected item in the list.
  selectedItemCount - the integer number of selected items.
  selectedItemPositions - a Collection containing the integer positions of the selected items.
  selectedItems - a Collection of items which are the selected items.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

Call data arguments:
  doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:
  item - the item which is the selected item.
  itemPosition - the integer position of the selected item in the list.
  selectedItemCount - the integer number of selected items.
  selectedItemPositions - a Collection containing the integer positions of the selected items.
  selectedItems - a Collection of items which are the selected items.

Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:
  item - the item which is the selected item.
  itemPosition - the integer position of the selected item in the list.
  selectedItemCount - the integer number of selected items.
  selectedItemPositions - a Collection containing the integer positions of the selected items.
  selectedItems - a Collection of items which are the selected items.

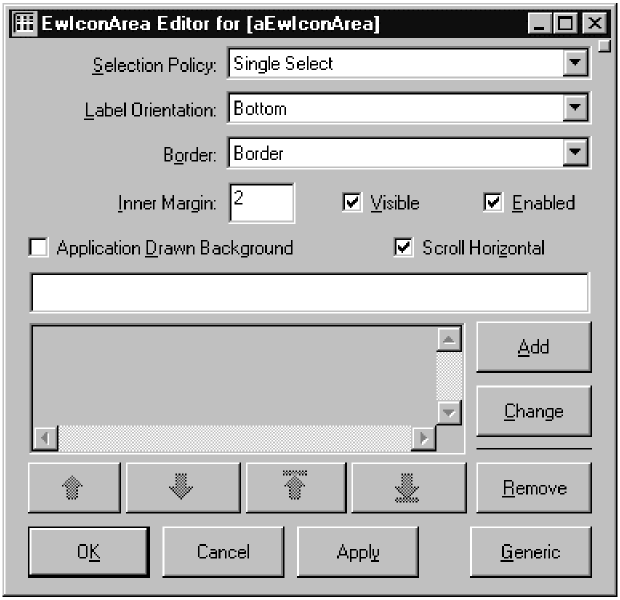
Visual Info Callback

These callbacks are triggered when an item’s icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
isInUse - the inUse status (Boolean) which is the default to be used for the item in the callback.
icon - the icon which is the default icon to be used for the item in the callback.
label - the label which is the default label to be used for the item in the callback.

Editor



EwIconArea Editor for [aEwIconArea]

Selection Policy: Single Select

Label Orientation: Bottom

Border: Border

Inner Margin: 2   ☑ Visible   ☑ Enabled

☐ Application Drawn Background   ☑ Scroll Horizontal

Application Drawn Background

Specifies whether the application wants the Draw Background Callback to fire when an item’s background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Inner Margin

Specifies the margin width to be used between each item's icon and its label.

Items

An array of objects that are to be displayed as the list items.

Label Orientation

Specifies the label position associated with the items.

Bottom - Position the label beneath the icon.
Right - Position the label to the right of the icon.

Scroll Horizontal

This resource specifies whether a horizontal scroll bar should be used for the list.

Selection Policy

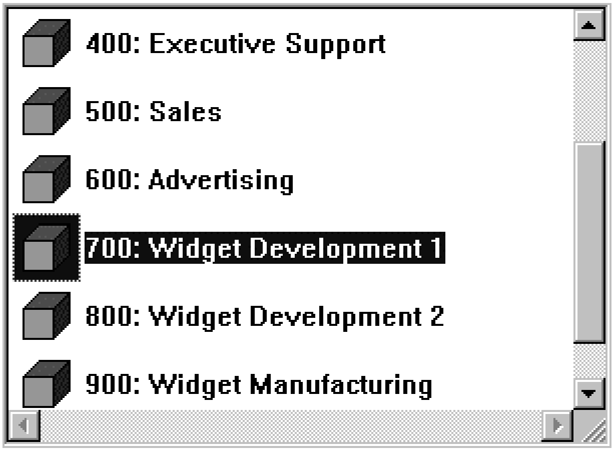
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.
Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
Read Only Select - Allows navigation, but no selection or callbacks.
Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwIconList


400: Executive Support
500: Sales
600: Advertising
700: Widget Development 1
800: Widget Development 2
900: Widget Manufacturing

This class provides a vertical, single-column list of items with an icon and a label for each item. The items in the list are typically actual application objects, not strings or arrays.

The widget then fires the visualInfo callback to obtain the label and icon for each item in the list as well as its inUse state. It only fires this callback for items which are visible on the screen. The application *must* hook the visualInfo callback and set the callData label to be the string (or other renderable object) which is to be the label for the item in callData item. The application *must* set the callData icon to be the CgIcon (or other renderable object) which is to be the icon for the item in callData item.

If the application desires a view with only a single icon or label per item, this can be achieved by setting the icon to nil for each item, setting the label to the desired visual for each item, setting the labelOrientation to XmRIGHT.

The objects used for item labels are typically Strings, and those used for item icons are typically CgIcons, but this is not a requirement. It is possible for an application to use any object as the label and icon for an item. Any item which understands `#ewHeightUsing`:, `#ewWidthUsing`: and `#ewDrawUsing`: (called a "renderable object") can be used for the label and icon. Since String and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

For example, if an application wishes to have different item's labels be different colors, it should implement a "ColoredString" class. This could then override the default rendering protocol and always render the string in a particular color, overriding the color in the gc of the EwRenderContext given. The same approach can be applied to handle item labels with different fonts, fixed alignments, etc.

Label editing can take place under program control only. When a label is about to be edited, the widget fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData `doit: true` to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the item label about to be edited. For example, if the item label contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the widget fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The label is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

emphasisPolicy: *anInteger*
Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Default: XmSEPARATE (Separate)
Valid resource values:

XmTOGETHER (Together) - Draw the icon and label emphasis as one single rectangle.

XmSEPARATE (Separate) - Draw the icon and label emphasis as two separate rectangles.

innerMargin: *anInteger*
Specifies the margin width to be used between each item's icon and its label.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

itemWidth: *anInteger*
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

labelOrientation: *anInteger*
Specifies the label position associated with the items.

Default: XmRIGHT (Right)
Valid resource values:
XmRIGHT (Right) - Position the label to the right of the icon.
XmBOTTOM (Bottom) - Position the label beneath the icon.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

Callbacks & Events

Begin Edit Callback

These callbacks are triggered when an item is about to be edited.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback

These callbacks are triggered when an item’s background needs to be drawn.

Call data arguments:
item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.

value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.
renderContext - the render context to be used in drawing the item's background
region - the region of the item whose background needs to be drawn.

End Edit Callback

These callbacks are triggered when an item is done being edited.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may 'undo' the selection change by setting the doit field of the callData to false.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

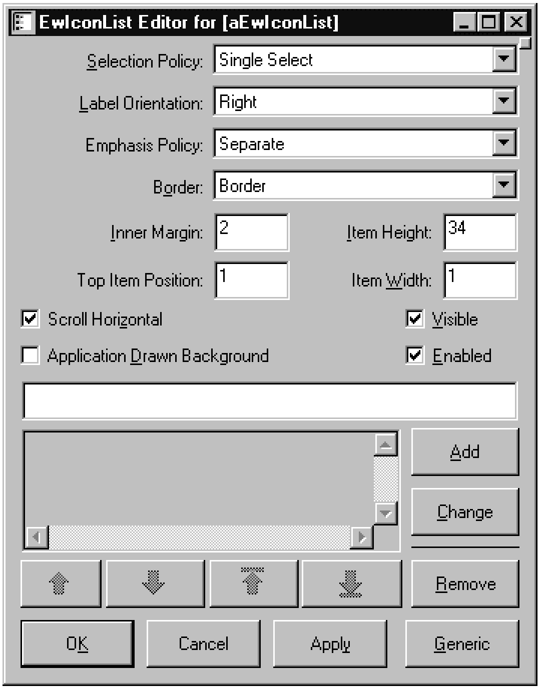
Visual Info Callback

These callbacks are triggered when an item's icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
isInUse - the inUse status (Boolean) which is the default to be used for the item in the callback.
icon - the icon which is the default icon to be used for the item in the callback.
label - the label which is the default label to be used for the item in the callback.

Editor



EwIconList Editor for [aEwIconList]

Selection Policy: Single Select

Label Orientation: Right

Emphasis Policy: Separate

Border: Border

Inner Margin: 2 Item Height: 34

Top Item Position: 1 Item Width: 1

☑ Scroll Horizontal ☑ Visible

□ Application Drawn Background ☑ Enabled

Application Drawn Background
Specifies whether the application wants the Draw Background Callback to fire when an item’s background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Emphasis Policy
Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Separate - Draw the icon and label emphasis as two separate rectangles.
Together - Draw the icon and label emphasis as one single rectangle.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Inner Margin
Specifies the margin width to be used between each item’s icon and its label.

Item Height
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Item Width
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

Items
An array of objects that are to be displayed as the list items.

Label Orientation
Specifies the label position associated with the items.

Bottom - Position the label beneath the icon.
Right - Position the label to the right of the icon.

Scroll Horizontal
This resource specifies whether a horizontal scroll bar should be used for the list.

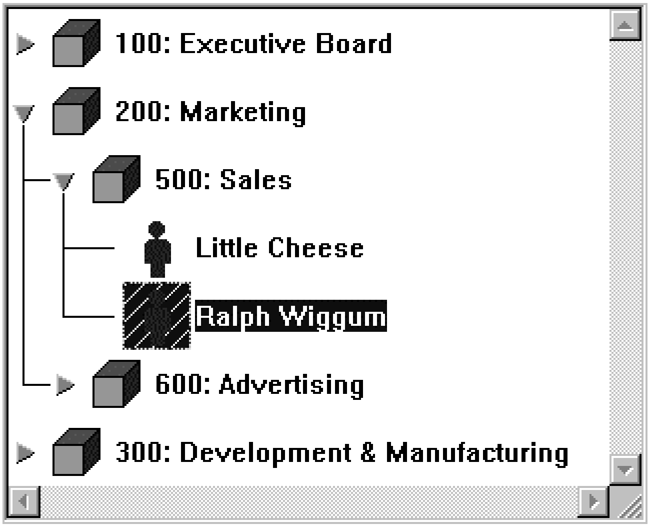
Selection Policy
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.
Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
Read Only Select - Allows navigation, but no selection or callbacks.
Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

Top Item Position
Specifies the Integer position of the item that is the first visible item in the list.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwIconTree



This class provides a vertical, hierarchical, single-column list of items with an icon and a label for each item. The items in the list are typically actual application objects, not strings or arrays. The application should only set the root level items in the tree as the items in the widget. The descendants of those root items can then be shown via `#expandPos:notify:` et al.

The hierarchical aspect of the list is handled by the visualInfoCallback and the childrenCallback. The application must hook the visualInfoCallback and set the callData `hasChildren` to true or false to indicate whether the item in callData item has children. The application must also hook the childrenCallback and set the callData `value` to be the list of children for the item in callData item. The childrenCallback will only ever fire for items for which hasChildren is true.

The hierarchyPolicy determines how the hierarchy is to be shown. This includes what the indentation level should be as well as whether to draw lines connecting the items and whether to show some kind of button beside items which have children. By default the hierarchyPolicy is an instance of EwHierarchyPolicy with lines set to true. The other hierarchy policy class provided is EwIconHierarchyPolicy. This class shows an icon beside each item to act as an expand/collapse button. The application can specify which icon to use in different situations so that the button can animate properly as it is pressed.

The widget then fires the visualInfo callback to obtain the label and icon for each item in the list as well as its inUse state. It only fires this callback for items which are visible on the screen. The application *must* hook the visualInfo callback and set the callData label to be the string (or other renderable object) which is to be the label for the item in callData item. The application *must* set the callData icon to be the CgIcon (or other renderable object) which is to be the icon for the item in callData item.

If the application desires a view with only a single icon or label per item, this can be achieved by setting the icon to nil for each item, setting the label to the desired visual for each item, setting the labelOrientation to XmRIGHT.

The objects used for item labels are typically Strings, and those used for item icons are typically CgIcons, but this is not a requirement. It is possible for an application to use any object as the label and icon for an item. Any item which understands `#ewHeightUsing`:, `#ewWidthUsing`: and `#ewDrawUsing`: (called a "renderable object") can be used for the label and icon. Since String and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

For example, if an application wishes to have different item's labels be different colors, it should implement a "ColoredString" class. This could then override the default rendering protocol and always render the string in a particular color, overriding the color in the gc of the EwRenderContext given. The same approach can be applied to handle item labels with different fonts, fixed alignments, etc.

Label editing can take place under program control only. When a label is about to be edited, the widget fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData `doit: true` to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the item label about to be edited. For example, if the item label contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the widget fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The label is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled* *window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

emphasisPolicy: *anInteger*
Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Default: XmSEPARATE (Separate)
Valid resource values:
XmTOGETHER (Together) - Draw the icon and label emphasis as one single rectangle.
XmSEPARATE (Separate) - Draw the icon and label emphasis as two separate rectangles.

hierarchyPolicy: *hierarchyPolicy*
Determines how the hierarchy is to be shown. This includes what the indentation level should be as well as whether to draw lines connecting the items and whether to show some kind of button beside items which have children. By default the hierarchyPolicy is an instance of EwHierarchyPolicy with lines set to true.

innerMargin: *anInteger*
Specifies the margin width to be used between each item's icon and its label.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

itemWidth: *anInteger*
Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

labelOrientation: *anInteger*
Specifies the label position associated with the items.

Default: XmRIGHT (Right)
Valid resource values:
XmRIGHT (Right) - Position the label to the right of the icon.
XmBOTTOM (Bottom) - Position the label beneath the icon.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.
XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

Callbacks & Events

Begin Edit Callback
These callbacks are triggered when an item is about to be edited.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.

editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Children Callback

These callbacks are triggered when an item's list of children is needed.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
children - the value of children for the item in the callback.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback

These callbacks are triggered when an item's background needs to be drawn.

Call data arguments:
item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.
value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.
renderContext - the render context to be used in drawing the item's background
region - the region of the item whose background needs to be drawn.

End Edit Callback

These callbacks are triggered when an item is done being edited.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Expand Collapse Callback

These callbacks are triggered when an item is expanded or collapsed.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

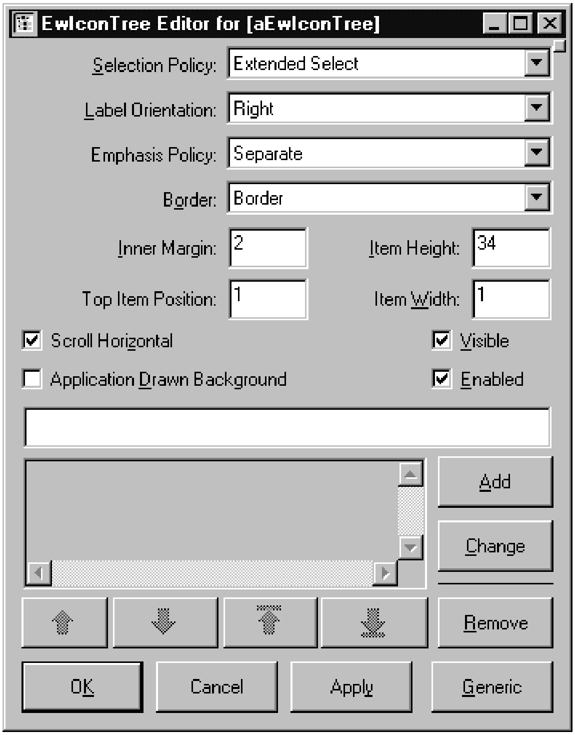
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.

selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Visual Info Callback
These callbacks are triggered when an item's icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
isInUse - the inUse status (Boolean) which is the default to be used for the item in the callback.
icon - the icon which is the default icon to be used for the item in the callback.
label - the label which is the default label to be used for the item in the callback.

Editor



Selection Policy: Extended Select
Label Orientation: Right
Emphasis Policy: Separate
Border: Border
Inner Margin: 2
Item Height: 34
Top Item Position: 1
Item Width: 1
☑ Scroll Horizontal
☑ Visible
☐ Application Drawn Background
☑ Enabled

Application Drawn Background

Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Emphasis Policy

Specifies whether to draw the icon and label emphasis as one single rectangle or as two separate rectangles.

Separate - Draw the icon and label emphasis as two separate rectangles.
Together - Draw the icon and label emphasis as one single rectangle.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Inner Margin

Specifies the margin width to be used between each item's icon and its label.

Item Height

Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Item Width

Specifies the width in pixels of items in the list. This includes the two pixels for emphasis.

Items

An array of objects that are to be displayed as the list items.

Label Orientation

Specifies the label position associated with the items.

Bottom - Position the label beneath the icon.
Right - Position the label to the right of the icon.

Scroll Horizontal

This resource specifies whether a horizontal scroll bar should be used for the list.

Selection Policy

Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Read Only Select - Allows navigation, but no selection or callbacks.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

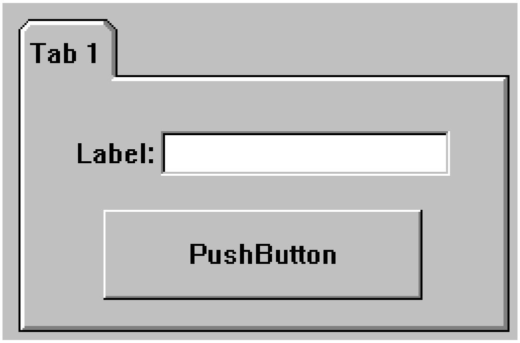
Top Item Position

Specifies the Integer position of the item that is the first visible item in the list.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwPage



A page is special type of composite that can only be added to a notebook. For layout of its children the page supports attachment protocol inherited from CwWidget. A page has the added functionality of being able to specify a drawable object for its tab, style of tab to use, and text for a status line. Although the last two options are only available when a page is a child of an OS/2 flavor notebook.

Callbacks are provided for finding out when a page is brought to the front of a notebook or when it is leaving the front of a notebook.

Protocol

fractionBase: *anInteger*
Specifies the denominator used in calculating the relative position a child widgets.

horizontalSpacing: *anInteger*
Specifies the offset for right and left attachments.

marginHeight: *anInteger*
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

marginWidth: *anInteger*
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

pageLabel: *aString*
Specifies the string to place in the status label.

resizePolicy: *anInteger*
Specifies the resize policy of the widget.

Default: XmRESIZEANY (Any)
Valid resource values:
XmRESIZENONE (None) - Resize none.
XmRESIZEGROW (Grow) - Resize grow.
XmRESIZEANY (Any) - Resize any.

rubberPositioning: *aBoolean*
Indicates the default attachment for a child of the Form. If this Boolean resource is set to false, then the left and top of the child defaults to being attached to the left and top side of the Form. If this resource is set to true, then the child defaults to being attached to its initial position in the Form.

tabLabel: *aString*
Specifies the renderable object to draw in the page's tab.

tabType: *anInteger*
Specifies the type of tab to use for the page.

Default: XmMAJOR (Major)
Valid resource values:
XmNONE (None) - These are major pages with no tabs.
XmMAJOR (Major) - These are the primary pages of a notebook.
XmMINOR (Minor) - These are essentially children of the major page they are added after. They display only after their parent is displayed and they appear perpendicular to the major tabs

verticalSpacing: *anInteger*
Specifies the offset for top and bottom attachments.

Callbacks & Events

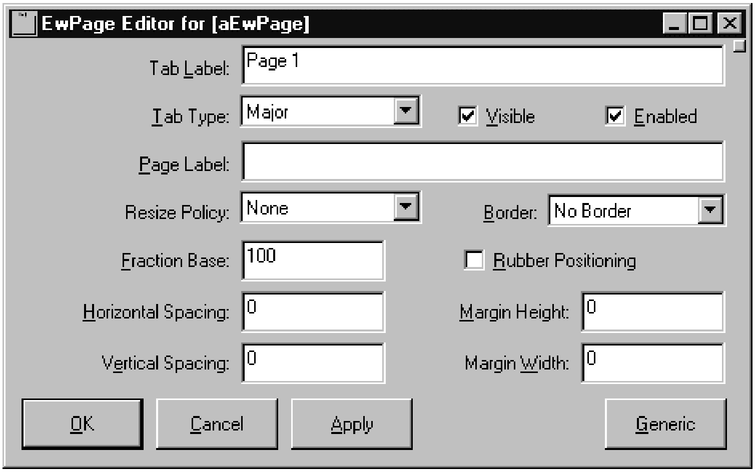
Page Enter Callback

These callbacks are triggered just before a page is to be managed. This happens when the receiver is being brought to the top in a parent notebook.

Page Leave Callback

These callbacks are triggered just before a page is to be unmanaged. This happens when another page is being brought to the top in a parent notebook

Editor



EwPage Editor for [aEwPage]

Tab Label: Page 1

Tab Type: Major ☑ Visible ☑ Enabled

Page Label:

Resize Policy: None Border: No Border

Fraction Base: 100 ☐ Rubber Positioning

Horizontal Spacing: 0 Margin Height: 0

Vertical Spacing: 0 Margin Width: 0

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Fraction Base
Specifies the denominator used in calculating the relative position a child widgets.

Horizontal Spacing
Specifies the offset for right and left attachments.

Margin Height
Specifies the minimum spacing in pixels between the top or bottom edge of the widget and any child widget.

Margin Width
Specifies the minimum spacing in pixels between the left or right edge of the widget and any child widget.

Page Label
Specifies the string to place in the status label.

Resize Policy
Specifies the resize policy of the widget.

Any - Resize any.
Grow - Resize grow.
None - Resize none.

Rubber Positioning
Indicates the default attachment for a child of the Form. If this Boolean resource is set to false, then the left and top of the child defaults to being attached to the left and top side of the Form. If this resource is set to true, then the child defaults to being attached to its initial position in the Form.

Tab Label
Specifies the renderable object to draw in the page's tab.

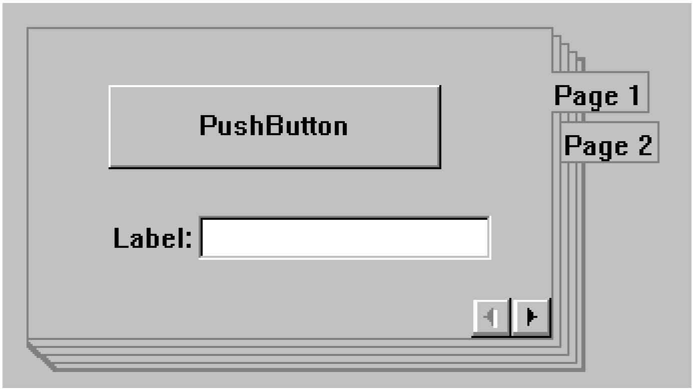
Tab Type
Specifies the type of tab to use for the page.

Major - These are the primary pages of a notebook.
Minor - These are essentially children of the major page they are added after. They display only after their parent is displayed and they appear perpendicular to the major tabs
None - These are major pages with no tabs.

Vertical Spacing
Specifies the offset for top and bottom attachments.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwPMNotebook



This class implements an OS/2 flavor notebook. The page's major/minor tabs can be displayed in different combinations on all sides of the notebook. The tabs' sizes are determined by the major and minor tabHeight, tabWidth, and tabWidthPolicy resources. The page on top which is displaying its widgets is represented by the currentPage resource.

The OS/2 notebook adds page buttons and a text area for each page. Page buttons allow the user to select the next or previous page. A text area, also called a status line, allows each page to display a single line of text relating to that page.

Protocol

backPagePosition: *anInteger*
Specifies where the simulated back pages appear.

Default: XmBOTTOMRIGHT (Bottom Right)
Valid resource values:
XmBOTTOMRIGHT (Bottom Right) - Back pages are drawn bottom and right.
XmTOPRIGHT (Top Right) - Back pages are drawn top and right.

bindingType: *anInteger*
Specifies the style of the binding.

Default: XmNONE (None)
Valid resource values:
XmNONE (None) - No binding is drawn.
XmSOLID (Solid) - A solid binding is drawn.
XmSPIRAL (Spiral) - A spiral pixmap is drawn.

majorTabHeight: *anInteger*
Specifies the height of the notebook’s major tabs in pixels.

majorTabWidth: *anInteger*
Specifies the width of the notebook’s major tabs in pixels.

minorTabHeight: *anInteger*
Specifies the height of the notebook’s minor tabs in pixels.

minorTabWidth: *anInteger*
Specifies the width of the notebook’s minor tabs in pixels.

orientation: *anInteger*
Specifies on what plane the pages turn. This is indicated by the major tabs being opposite the binding.

Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - Major tabs are either on the top or bottom depending on the back page position and minor tabs appear on the right.
XmHORIZONTAL (Horizontal) - Major tabs appear on the right and minor tabs are either on the top or bottom depending on the back page position.

pageButtonHeight: *anInteger*
Specifies the height of the notebook’s page buttons in pixels. A value of zero means the are no buttons or page label visible.

pageButtonWidth: *anInteger*
Specifies the width of the notebook’s page buttons in pixels. A value of zero means that only the page buttons are not visible.

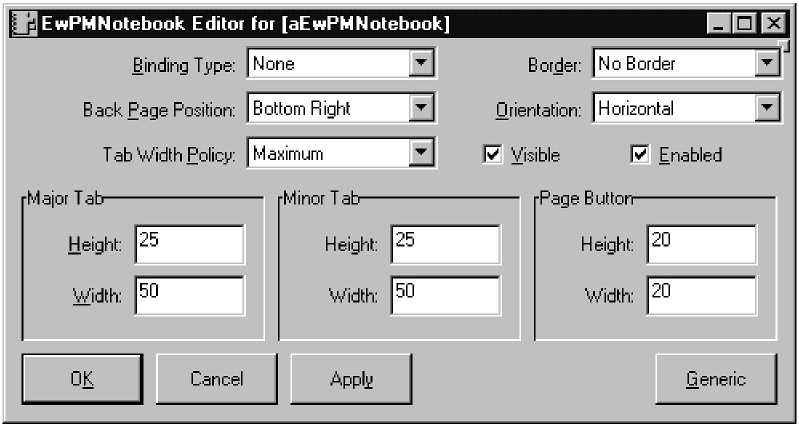
tabWidthPolicy: *anInteger*
Specifies the technique that will be used to set the width of the tabs in a notebook.

Default: XmMAXIMUM (Maximum)
Valid resource values:
XmCONSTANT (Constant) - The tabs will be sized according to the value of the majorTabWidth and minorTabWidth resources.
XmVARIABLE (Variable) - The tabs will be individually sized to fit their labels.
XmMAXIMUM (Maximum) - The tabs will all be the size of the tab needed to accomodate the widest tab label.

Callbacks & Events

Page Change Callback
These callbacks are triggered just before any switching of pages take place.

Editor



EwPMNotebook Editor for [aEwPMNotebook]

Binding Type: None

Border: No Border

Back Page Position: Bottom Right

Orientation: Horizontal

Tab Width Policy: Maximum

☑ Visible ☑ Enabled

Major Tab
Height: 25
Width: 50

Minor Tab
Height: 25
Width: 50

Page Button
Height: 20
Width: 20

Back Page Position
Specifies where the simulated back pages appear.

> Bottom Right - Back pages are drawn bottom and right.
> Top Right - Back pages are drawn top and right.

Binding Type
Specifies the style of the binding.

> None - No binding is drawn.
> Solid - A solid binding is drawn.
> Spiral - A spiral pixmap is drawn.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

> Border - Causes the widget to have a border.
> No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Major Tab Height
Specifies the height of the notebook's major tabs in pixels.

Major Tab Width
Specifies the width of the notebook's major tabs in pixels.

Minor Tab Height
Specifies the height of the notebook’s minor tabs in pixels.

Minor Tab Width
Specifies the width of the notebook’s minor tabs in pixels.

Orientation
Specifies on what plane the pages turn. This is indicated by the major tabs being opposite the binding.

Horizontal - Major tabs appear on the right and minor tabs are either on the top or bottom depending on the back page position.
Vertical - Major tabs are either on the top or bottom depending on the back page position and minor tabs appear on the right.

Page Button Height
Specifies the height of the notebook’s page buttons in pixels. A value of zero means the are no buttons or page label visible.

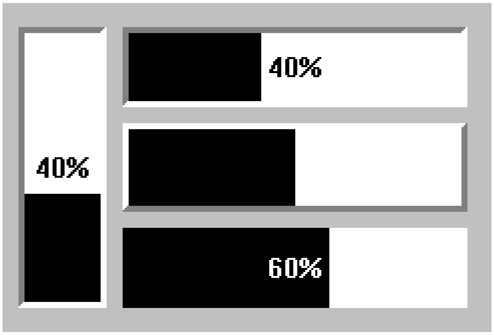
Page Button Width
Specifies the width of the notebook’s page buttons in pixels. A value of zero means that only the page buttons are not visible.

Tab Width Policy
Specifies the technique that will be used to set the width of the tabs in a notebook.

Constant - The tabs will be sized according to the value of the majorTabWidth and minorTabWidth resources.
Variable - The tabs will be individually sized to fit their labels.
Maximum - The tabs will all be the size of the tab needed to accommodate the widest tab label.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwProgressBar



EwProgressBar provides a visual representation of a tasks progress, as a rectangular area whose size represents a fraction of a total task.

The progress bar can be displayed horizontally and vertically, and the direction of the progress can be specified (left-right, right-left, bottom-top, top-bottom). The size of the progress bar's completion indication is controlled by the XmNfractionComplete resource. This resource is specified as a number (typically a fraction) between 0 and 1. 0 represents no progress, and 1 represents completion. The progress bar's completion indication can either be a color (XmNforegroundColor resource) or any renderable image (such as an icon or bitmap -- XmNribbonImage resource). The uncompleted progress indication is a color (XmNbackgroundColor resource). A label can be drawn centered in the progress bar. This can either be a percentage complete, displayed automatically by the progress bar (XmNshowPercentage resource), or any defined renderable image as the label (XmNimage resource).

Protocol

direction: *anInteger*
Specifies the direction the progress bar moves in.

Default: XmFORWARD (Forward)
Valid resource values:

XmFORWARD (Forward) - The progress bar moves forward. For horizontal progress bars, this is left-to-right. For vertical progress bars, this is top-to-bottom.

XmREVERSE (Reverse) - The progress bar moves backwards. For horizontal progress bars, this is right-to-left. For vertical progress bars, this is bottom-to-top.

fractionComplete: *anInteger*
Specifies the current amount of progress to show in the progress bar. This resource is a fraction, denoting a number between 0 and 1. For example, 1/10 specifies 10 % complete. 1 represents 100 % complete.

image: *aPixmap*
Specifies the renderable object which draws as the label of the progress bar. The label is displayed only if the showPercentage resource value is false. Note that the image label is drawn twice, in the foreground and background colors, so that the label appears 'reversed' over the actual foreground and background colors of the progress bar. If a ribbon image is being used, then the label is drawn once, using the foreground color.

imageColor: *aCgRGBColor*
Specifies the color of the label drawn according to the image resource, or the color of the percentage complete label, according to the showPercentage resource. If the resource value is nil, then the label is drawn in reverse color over the colors of the progress bar (foreground color) and the background color.

orientation: *anInteger*
Specifies the orientation of the progress bar.
Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - Display the progress bar vertically.
XmHORIZONTAL (Horizontal) - Display the progress bar horizontally.

ribbonImage: *aPixmap*
Specifies the renderable object which draws as the completed ribbon of the progress bar. This object is drawn instead of a color strip. Note that when a ribbon image is used, a label (the showPercentage or image resources) will be drawn using the imageColor resource, or if that is nil, the foregroundColor resource.

shadowType: *anInteger*
Specifies the drawing style for the frame around the progress bar widget.
Default: XmSHADOWIN (Shadow In)
Valid resource values:
XmSHADOWNONE (Shadow None) - No frame is drawn.
XmSHADOWIN (Shadow In) - Draws a frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
XmSHADOWOUT (Shadow Out) - Draws a frame such that it appears outset.

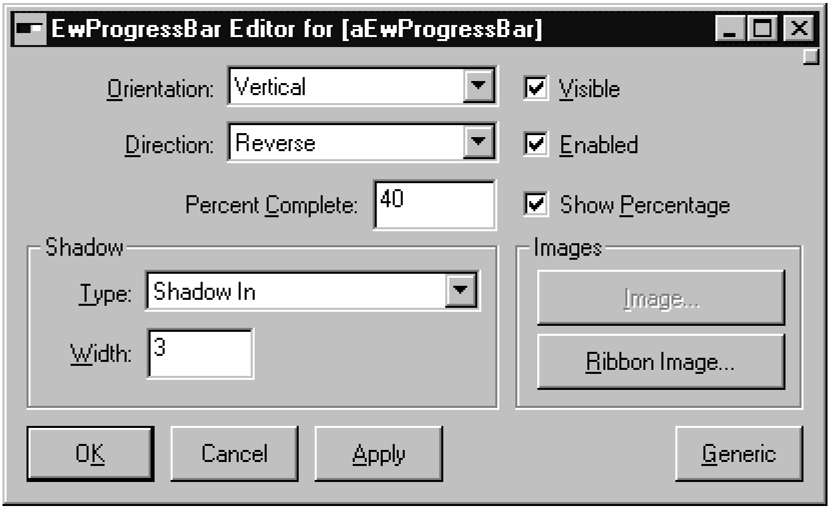
shadowWidth: *anInteger*
Specifies the width for the border

showPercentage: *aBoolean*
Specifies the whether a label showing the percentage completed is shown in the progress bar. If true, then the string 'X %' is show in the progress bar, where X is the percentage of progress completed. If false, then no percentage label is shown.

Callbacks & Events

None

Editor



EwProgressBar Editor for [aEwProgressBar]

Orientation: Vertical

Direction: Reverse

Percent Complete: 40

☑ Visible

☑ Enabled

☑ Show Percentage

Shadow

Type: Shadow In

Width: 3

Images

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Direction
Specifies the direction the progress bar moves in.

Forward - The progress bar moves forward. For horizontal progress bars, this is left-to-right. For vertical progress bars, this is top-to-bottom.
Reverse - The progress bar moves backwards. For horizontal progress bars, this is right-to-left. For vertical progress bars, this is bottom-to-top.

Percent Complete
Specifies the current amount of progress to show in the progress bar. This should be a number between 0 and 100.

Image
Specifies the renderable object which draws as the label of the progress bar. The label is displayed only if the showPercentage resource value is false. Note that the image label is drawn twice, in the foreground and background colors, so that the label appears ‘reversed’ over the actual foreground and background colors of the progress bar. If a ribbon image is being used, then the label is drawn once, using the foreground color.

Orientation

Specifies the orientation of the progress bar.

Vertical - Display the progress bar vertically.
Horizontal - Display the progress bar horizontally.

Ribbon Image

Specifies the renderable object which draws as the completed ribbon of the progress bar. This object is drawn instead of a color strip. Note that when a ribbon image is used, a label (the showPercentage or image resources) will be drawn using the imageColor resource, or if that is nil, the foregroundColor resource.

Shadow Type

Specifies the drawing style for the frame around the progress bar widget.

Shadow None - No frame is drawn.
Shadow In - Draws a frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
Shadow Out - Draws a frame such that it appears outset.

Shadow Width

Specifies the width for the border

Show Percentage

Specifies the whether a label showing the percentage completed is shown in the progress bar. If true, then the string ‘X %’ is show in the progress bar, where X is the percentage of progress completed. If false, then no percentage label is shown.

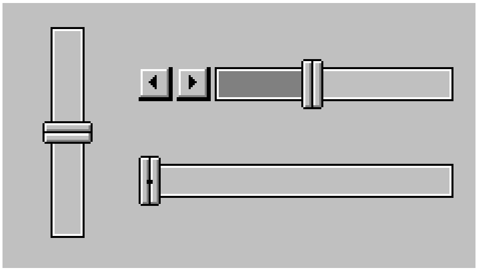
Value

Specifies the slider current position along the scale, between minimum and maximum.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwSlider



EwSlider control provides an analog representation of a value within a specified range. The range is represented by a horizontal or vertical shaft. The current value is indicated by the position of a slider arm which can be moved along the length of the shaft.

EwSlider provides functionality for two scales, which may be displayed above and below a horizontal slider, or to the left and right of a vertical slider. Both scales provide a minimum value, maximum value and resolution. The resolution represents the size of the increments between the minimum and maximum, (e.g. min = 0, max = 100, resolution = 10 would result in valid values of 0, 10, 20...100). Although it is not necessary to display any tick marks, a primary scale must be specified. A secondary scale is optional.

EwSlider also provides the option of increment and decrement buttons. If requested, these buttons can be positioned together at either end of the shaft, or separately, one at each end of the shaft. The slider value is changed by moving the slider arm; either by dragging it with the mouse, depressing the increment or decrement buttons, or by using the arrow keys on the keyboard. The slider arm moves in units specified by the resolution of the current scale, as specified by the xmNcurrentScale resource.

Protocol

activeScale: *anInteger*
Indicates which scale is the active scale. The active scale indicates the scale to be used when positioning the slider arm.

Default: XmTOPORLEFT (Top or Left)
Valid resource values:
XmTOPORLEFT (Top or Left) - Use the top or left scale (depending on the value of the orientation resource) when manipulating the slider value.
XmBOTTOMORRIGHT (Bottom or Right) - Use the bottom or right scale (depending on the value of the orientation resource) when manipulating the slider value.

bottomOrRightScaleMax: *anInteger*
Specifies the slider's maximum value for the bottom or right scale.

bottomOrRightScaleMin: *anInteger*
Specifies the slider's minimum value for the bottom or right scale.

bottomOrRightScaleResolution: *anInteger*
Represents the size of the increments between the min and max values for the bottom or right scale, e.g., a resolution of 2 for a scale with min = 0 and max = 100 would result in 51 increments, namely 0, 2, 4, 6,...,100.

bottomOrRightScaleShaftIncrement: *anInteger*
Represents the amount to be added to or subtracted from the current value when the left mouse button is pressed inside the shaft to the left or right of the slider arm, respectively. The value of the shaftIncrement resource must be a multiple of the scaleResolution resource.

bottomOrRightScaleValue: *anInteger*
Specifies the value associated with the slider's current position along the bottom or right scale, between minimum and maximum.

buttonStyle: *anInteger*
Indicates where buttons should be displayed relative to the shaft.

Default: XmBUTTONSNONE (No Buttons)
Valid resource values:
XmBUTTONSNONE (No Buttons) - No buttons.
XmBUTTONSSPLIT (Split) - One button on either end of the shaft.
XmBUTTONSBEGINNING (Beginning) - For vertical sliders, both buttons above shaft; for horizontal sliders, both buttons to the left of the shaft.
XmBUTTONSEND (End) - For vertical sliders, both buttons below shaft; for horizontal sliders, both buttons to the right of the shaft.

horizontalMargin: *anInteger*
Indicates the number of pixels to be used as a margin between the slider components (scales, shaft, buttons) and the left and right edges of the widget's area.

orientation: *anInteger*
Indicates whether slider should be displayed vertically or horizontally.

Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - Display the slider vertically.
XmHORIZONTAL (Horizontal) - Display the slider horizontally.

readOnly: *aBoolean*
Indicates whether the slider is being used as a read-only status indicator. If so, the value of the ribbonStrip resource is set to true and no slider arm or buttons are displayed.

ribbonStrip: *aBoolean*
Indicates whether the area between the minimum and the slider arm should be filled.

snapToResolution: *aBoolean*
Indicates the current snap policy, which regulates the positioning of the slider arm when the mouse button is released after dragging. The slider's current value will always be a multiple of the resolution, regardless of the snap policy.

thickness: *anInteger*
Specifies the slider shaft vertical thickness (in pixels) for a horizontal slider or horizontal thickness (in pixels) for a vertical slider.

topOrLeftScaleMax: *anInteger*
Specifies the slider's maximum value for the top or left scale.

topOrLeftScaleMin: *anInteger*
Specifies the slider's minimum value for the top or left scale.

topOrLeftScaleResolution: *anInteger*
Represents the size of the increments between the min and max values for the top or left scale, e.g., a resolution of 2 for a scale with min = 0 and max = 100 would result in 51 increments, namely 0, 2, 4, 6,...,100.

topOrLeftScaleShaftIncrement: *anInteger*
Represents the amount to be added to or subtracted from the current value when the left mouse button is pressed inside the shaft to the left or right of the slider arm, respectively. The value of the shaftIncrement resource must be a multiple of the scaleResolution resource.

topOrLeftScaleValue: *anInteger*
Specifies the value associated with the slider's current position along the top or left scale, between minimum and maximum.

verticalMargin: *anInteger*
Indicates the number of pixels to be used as a margin between the slider components (scales, shaft, buttons) and the top and bottom edges of the widget's area.

Callbacks & Events

Drag Callback

These callbacks are triggered when the slider position changes as the arm is being dragged.

Call data arguments:

topOrLeftScaleValue - the value associated with the top or left scale.
bottomOrRightScaleValue - the value associated with the bottom or right scale.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback

These callbacks are triggered before the widget loses input focus.

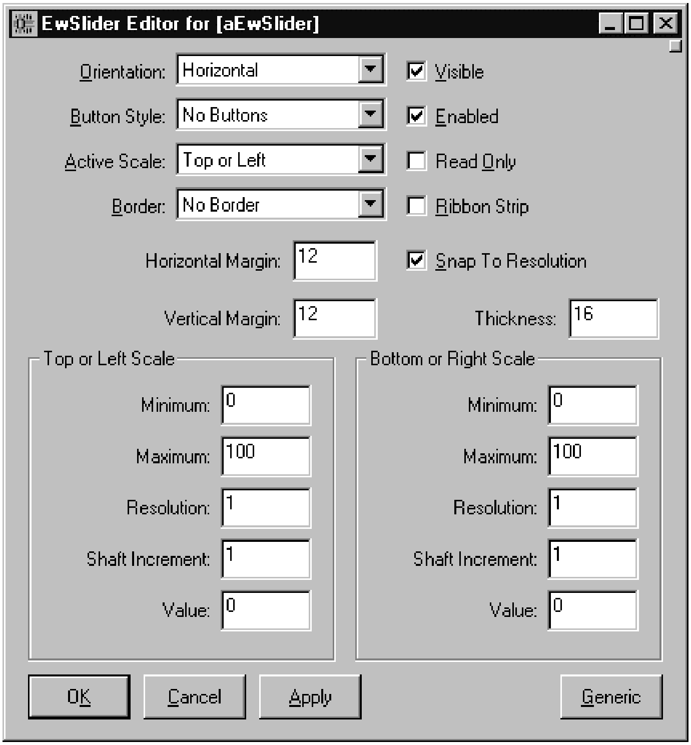
Value Changed Callback

These callbacks are triggered when the value of the slider has changed.

Call data arguments:

topOrLeftScaleValue - the value associated with the top or left scale.
bottomOrRightScaleValue - the value associated with the bottom or right scale.

Editor



EwSlider Editor for [aEwSlider]

Orientation: Horizontal
Button Style: No Buttons
Active Scale: Top or Left
Border: No Border

☑ Visible
☑ Enabled
□ Read Only
□ Ribbon Strip
☑ Snap To Resolution

Horizontal Margin: 12
Vertical Margin: 12
Thickness: 16

Top or Left Scale		Bottom or Right Scale	
Minimum:	0	Minimum:	0
Maximum:	100	Maximum:	100
Resolution:	1	Resolution:	1
Shaft Increment:	1	Shaft Increment:	1
Value:	0	Value:	0

Active Scale
Indicates which scale is the active scale. The active scale indicates the scale to be used when positioning the slider arm.

Bottom or Right - Use the bottom or right scale (depending on the value of the orientation resource) when manipulating the slider value.
Top or Left - Use the top or left scale (depending on the value of the orientation resource) when manipulating the slider value.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Bottom Or Right Scale Max
Specifies the slider's maximum value for the bottom or right scale.

Bottom Or Right Scale Min
Specifies the slider's minimum value for the bottom or right scale.

Bottom Or Right Scale Resolution
Represents the size of the increments between the min and max values for the bottom or right scale, e.g., a resolution of 2 for a scale with min = 0 and max = 100 would result in 51 increments, namely 0, 2, 4, 6,...,100.

Bottom Or Right Scale Shaft Increment
Represents the amount to be added to or subtracted from the current value when the left mouse button is pressed inside the shaft to the left or right of the slider arm, respectively. The value of the shaftIncrement resource must be a multiple of the scaleResolution resource.

Bottom Or Right Scale Value
Specifies the value associated with the slider's current position along the bottom or right scale, between minimum and maximum.

Button Style
Indicates where buttons should be displayed relative to the shaft.

Beginning - For vertical sliders, both buttons above shaft; for horizontal sliders, both buttons to the left of the shaft.
End - For vertical sliders, both buttons below shaft; for horizontal sliders, both buttons to the right of the shaft.
No Buttons - No buttons.
Split - One button on either end of the shaft.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Horizontal Margin
Indicates the number of pixels to be used as a margin between the slider components (scales, shaft, buttons) and the left and right edges of the widget's area.

Orientation
Indicates whether slider should be displayed vertically or horizontally.

Horizontal - Display the slider horizontally.
Vertical - Display the slider vertically.

Read Only
Indicates whether the slider is being used as a read-only status indicator. If so, the value of the ribbonStrip resource is set to true and no slider arm or buttons are displayed.

Ribbon Strip
Indicates whether the area between the minimum and the slider arm should be filled.

Snap To Resolution
Indicates the current snap policy, which regulates the positioning of the slider arm when the mouse button is released after dragging. The slider's current value will always be a multiple of the resolution, regardless of the snap policy.

Thickness
Specifies the slider shaft vertical thickness (in pixels) for a horizontal slider or horizontal thickness (in pixels) for a vertical slider.

Top Or Left Scale Max
Specifies the slider's maximum value for the top or left scale.

Top Or Left Scale Min
Specifies the slider's minimum value for the top or left scale.

Top Or Left Scale Resolution
Represents the size of the increments between the min and max values for the top or left scale, e.g., a resolution of 2 for a scale with min = 0 and max = 100 would result in 51 increments, namely 0, 2, 4, 6,...,100.

Top Or Left Scale Shaft Increment
Represents the amount to be added to or subtracted from the current value when the left mouse button is pressed inside the shaft to the left or right of the slider arm, respectively. The value of the shaftIncrement resource must be a multiple of the scaleResolution resource.

Top Or Left Scale Value

Specifies the value associated with the slider’s current position along the top or left scale, between minimum and maximum.

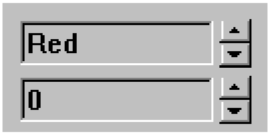
Vertical Margin

Indicates the number of pixels to be used as a margin between the slider components (scales, shaft, buttons) and the top and bottom edges of the widget’s area.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

EwSpinButton


Red

0

A spin button has three parts. An entry field which displays one value from a range of numeric values or a collection of string values. And up and down arrow buttons which allow the user to scroll though the values. A user can increment or decrement the values using up/down arrow keys or by selecting the up/down buttons. A user can change the value in an editable spin button by typing in the entry field.

Protocol

editable: *aBoolean*
Specifies whether a user can edit text in the entry field part of the spin button.

increment: *anInteger*
Specifies the amount to increase or decrease a numeric spin button when the corresponding arrow is selected.

items: *anOrderedCollection*
Specifies the list of Strings being spun over.

maximum: *anInteger*
Specifies a numeric spin button's maximum value.

minimum: *anInteger*
Specifies a numeric spin button's minimum value.

wrap: *aBoolean*
Specifies whether the spin button should cycle or stop upon reaching the end of the collection, or max or min for a numeric spin button.

Callbacks & Events

Activate Callback

These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Decrement Callback

These callbacks are triggered when the user decreases the spin button value by one step. This can happen either when the down arrow key is pressed or when the down arrow button is selected with the mouse.

Focus Callback

These callbacks are triggered before the entry field has accepted input focus.

Increment Callback

These callbacks are triggered when the user increases the spin button value by one step. This can happen either when the up arrow key is pressed or the up arrow button is selected with the mouse.

Losing Focus Callback

These callbacks are triggered before the entry field loses input focus.

Modify Verify Callback

These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

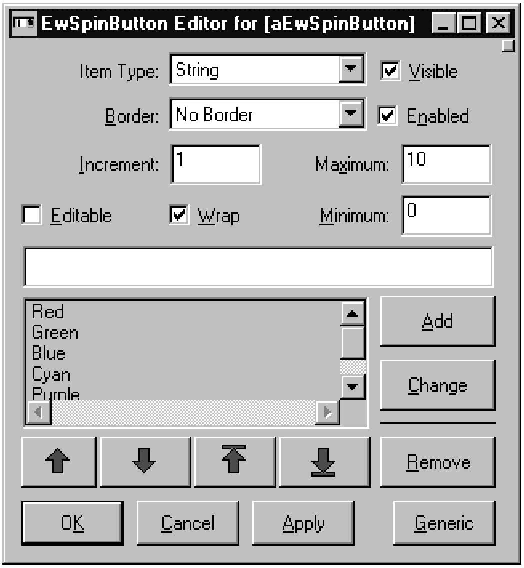
Call data arguments:

doit - Indicates whether the action that invoked the callback is performed.
Setting doit to false negates the action.
text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

Value Changed Callback

These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



EwSpinButton Editor for [aEwSpinButton]

Item Type: String

☑ Visible

Border: No Border

☑ Enabled

Increment: 1

Maximum: 10

☐ Editable

☑ Wrap

Minimum: 0

Red
Green
Blue
Cyan
Purple

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Editable
Specifies whether a user can edit text in the entry field part of the spin button.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Increment
Specifies the amount to increase or decrease a numeric spin button when the corresponding arrow is selected.

Items
Specifies the list of Strings being spun over.

Item Type
Specifies whether the spin button spins through a set of numbers or stings

Numeric - Spin through a set of numbers.
String - Spin through a set of strings.

Maximum
Specifies a numeric spin button’s maximum value.

Minimum
Specifies a numeric spin button’s minimum value.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Wrap
Specifies whether the spin button should cycle or stop upon reaching the end of the collection, or max or min for a numeric spin button.

EwTableColumn

The EwTableColumn class defines a single column in an EwTableList or EwTableTree widget. In addition to details about the look and feel of the column, it also provides a callback which the application must hook to provide the cell value for an item in that column. The column also provides callbacks to notify the application when a cell is about to be edited and when the edit is about to end.

Note: EwTableColumn objects may only be created from within the EwTableList or ExTableTree editors.

Protocol

editable: *aBoolean*
Specifies whether the cells in this column are editable.

etched: *aBoolean*
Specifies whether this column is to be etched.

heading: *aString*
Specifies the heading object to be displayed at the top of the column.

horizontalAlignment: *anInteger*
Specifies how the cells in this column should be aligned horizontally.

Default: XmALIGNMENTBEGINNING (Left)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Align the cells to the left.
XmALIGNMENTCENTER (Center) - Align the cells to the center.
XmALIGNMENTEND (Right) - Align the cells to the right.

horizontalHeadingAlignment: *anInteger*
Specifies how this column’s heading should be aligned horizontally.

Default: XmALIGNMENTBEGINNING (Left)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Align the heading to the left.
XmALIGNMENTCENTER (Center) - Align the heading to the center.
XmALIGNMENTEND (Right) - Align the heading to the right.

resizable: *aBoolean*
Specifies whether the column is resizable by the user. If so, the user can drag the right side of the column heading to change the column width

showInUse: *aBoolean*
Specifies whether the column should show the inUse emphasis for each item.

verticalAlignment: *anInteger*
Specifies how the cells and heading in this column should be aligned vertically.

Default: XmALIGNMENTBEGINNING (Top)
Valid resource values:
    XmALIGNMENTBEGINNING (Top) - Align the cells to the top.
    XmALIGNMENTCENTER (Center) - Align the cells to the vertical center.
    XmALIGNMENTEND (Bottom) - Align the cells to the bottom.

verticalSeparatorThickness: *anInteger*
Specifies the thickness of the line shown to the right of each cell in the column.

width: *anInteger*
Specifies the width of the column in pixels. This does not includes the width of any emphasis or vertical separator.

Callbacks & Events

Begin Edit Callback

These callbacks are triggered when an item is about to be edited.

Call data arguments:
    item - the item which is the selected item.
    itemPosition - the integer position of the selected item in the list.
    doit - indicates whether the action that invoked the callback is performed.
    editPolicy - the edit policy to be used to edit the cell value.
    value - the value of the cell which is about to be edited.

Cell Value Callback

These callbacks are triggered when an item’s cell value is needed for this column.

Call data arguments:
    item - the item which is the selected item.
    itemPosition - the integer position of the selected item in the list.
    column - the EwTableColumn for which a cell value is needed.
    value - the value of the cell which is about to be edited.

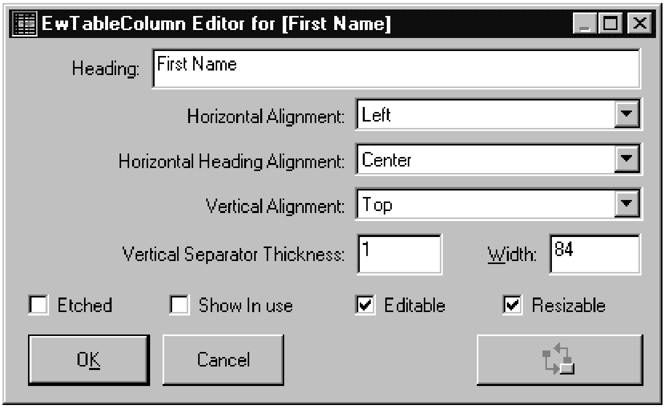
End Edit Callback

These callbacks are triggered when an item is done being edited.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
doit - indicates whether the action that invoked the callback is performed.
editPolicy - the edit policy to be used to edit the cell value.
value - the value of the cell which is about to be edited.

Editor



EwTableColumn Editor for [First Name]

Heading: First Name
Horizontal Alignment: Left
Horizontal Heading Alignment: Center
Vertical Alignment: Top
Vertical Separator Thickness: 1 Width: 84
☐ Etched ☐ Show In use ☑ Editable ☑ Resizable

Callbacks
Launches the Callback Editor on the edited column.

Editable

Specifies whether the cells in this column are editable.

Etched

Specifies whether this column is to be etched.

Heading

Specifies the heading object to be displayed at the top of the column.

Horizontal Alignment

Specifies how the cells in this column should be aligned horizontally.

Center - Align the cells to the center.
Left - Align the cells to the left.
Right - Align the cells to the right.

Horizontal Heading Alignment

Specifies how this column’s heading should be aligned horizontally.

Center - Align the heading to the center.
Left - Align the heading to the left.
Right - Align the heading to the right.

Resizable

Specifies whether the column is resizable by the user. If so, the user can drag the right side of the column heading to change the column width

Show In Use

Specifies whether the column should show the inUse emphasis for each item.

Vertical Alignment

Specifies how the cells and heading in this column should be aligned vertically.

Bottom - Align the cells to the bottom.
Center - Align the cells to the vertical center.
Top - Align the cells to the top.

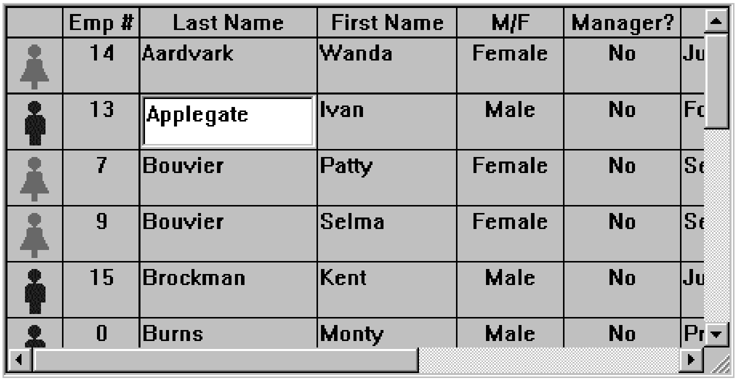
Vertical Separator Thickness

Specifies the thickness of the line shown to the right of each cell in the column.

Width

Specifies the width of the column in pixels. This does not includes the width of any emphasis or vertical separator.

EwTableList



	Emp #	Last Name	First Name	M/F	Manager?	
	14	Aardvark	Wanda	Female	No	Ju
	13	Applegate	Ivan	Male	No	Fc
	7	Bouvier	Patty	Female	No	Se
	9	Bouvier	Selma	Female	No	Se
	15	Brockman	Kent	Male	No	Ju
	0	Burns	Monty	Male	No	Pr

EwTableList provides a vertical, multi-column list of items which shows a row of cells for each item in the list. The items in the list are typically actual application objects, not strings or arrays. For example, the items (rows) might be employee objects, and the columns might show the name, salary, and hire date for each employee.

The application must define the columns in the table by providing an array of EwTableColumn instances. These each define a single column in the table. In addition to defining the heading, width and other visual parameters, the column also provides a cellValueCallback. The application *must* hook this callback on each column. This callback fires whenever the widget needs to know what value to put in a cell for a given item. The application must then set the callData value to be the renderable object (see below) which is to reside in the cell. For example, when the widget needs to place an employee's salary in the salary column, it fires the callValueCallback. The callData tells for which employee the salary is needed (callData item). The application might hook the cellValueCallback from the Salary column to a method like:

```
salary: aWidget clientData: clientData callData: callData
   "The widget needs an employee's salary."
   | employee |
   employee := callData item.
   callData value: employee salary.
```

The application may or may not hook the headingCellValueCallback. Column headings can be set via the EwTableColumn>>heading: method or via the headingCellValueCallback.

Both the headingCellValueCallback and cellValueCallback can be used to set color for a particular cell. Colors can be specified for the foreground, background, selectedForeground, and selectedBackground of a cell.

The objects used for cell values and headings are typically Strings, Numbers, and CgIcons, but this is not a requirement. It is possible for an application to use any object as the cell value or heading. Any item which understands `#ewHeightUsing`:, `#ewWidthUsing`: and `#ewDrawUsing`: (called a "renderable object") can be used for a cell value. Since String, Number, and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

Cell editing can take place automatically or under program control. For cell editing to be automatic, the widget must have editable true, and the columns in which cells may be edited must also have editable true. Under program control, this is not required.

When a cell is about to be edited, either because the user clicked in it or because the program called `#editCellAt`: or `#editSelectedCell`, the column fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData doit: true to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the cell about to be edited. For example, if the cell contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the column fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The cell is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an

item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

cellTabbingPolicy: *anInteger*
Specifies how tabbing should occur amongst cells.
Default: XmACROSSROWS (Across Rows)
Valid resource values:

XmACROSSROWS (Across Rows) - When the last cell in a row is reached when tabbing forward, go to the first cell in the next row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the previous row.

XmAPPLICATIONDEFINED (Application Defined) - Tabbing behavior is defined by the application. To use this option, the application must hook the Cell Tab Callback.

XmWITHINROW (Within Row) - When the last cell in a row is reached when tabbing forward, go to the first cell in the same row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the same row.

editable: *aBoolean*
Specifies whether the cells in the table are editable. For a cell to be editable, the table widget must be editable, the column must be editable, and the application must hook the Begin Edit Callback and set the callData doit to true

headingFont: *aString*
Specifies the font associated with the headings.

headingSeparatorThickness: *anInteger*
Specifies the thickness of the horizontal line separating the column headings from the rest of the table.

headingVisualStyle: *anInteger*
Defines the how the cells are displayed.
Default: XmETCHCELLS (Etch Cells)
Valid resource values:

XmFLAT (Flat) - Looks like a list box.

XmETCHROWS (Etch Rows) - Each row is etched.

XmETCHCELLS (Etch Cells) - Each cell in an etched column is individually etched.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

lockedColumns: *anInteger*
Specifies the number of columns to be locked down on the left side of the table. A column can be locked only if its preceding column is also locked. When scrolling, the

locked columns remain fixed on the left side and all other columns scroll under the locked columns. This value must be no greater than the number of columns.

rowSeparators: *aBoolean*
Specifies whether the rows are separated by a horizontal line.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectableColumns: *aBoolean*
Defines whether the user may select columns by clicking on their headings.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:

XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select

XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.

XmCELLSINGLESELECT (Cell Single Select) - Allows single selection of cells.

XmCELLBLOCKSELECT (Cell Block Select) - Allows selection of a rectangular block of cells.

separatorsToExtremes: *aBoolean*
Defines whether row and column separators are to be extended to the extreme right and bottom edges of the table.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

visualStyle: *anInteger*
Defines the how the cells are displayed.

Default: XmFLAT (Flat)
Valid resource values:
XmFLAT (Flat) - Looks like a list box.
XmETCHROWS (Etch Rows) - Each row is etched.
XmETCHCELLS (Etch Cells) - Each cell in an etched column is individually etched.

Callbacks & Events

Browse Selection Callback
These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Cell Block Selection Callback
These callbacks are triggered when a block of cells is selected in cell block selection mode.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.
columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Cell Single Selection Callback
These callbacks are triggered when an item is selected in cell single selection mode.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Column Heading Selection Callback
These callbacks are triggered when a column heading is selected

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.
columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Default Action Callback
These callbacks are triggered when an item is double clicked.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback
These callbacks are triggered when an item’s background needs to be drawn.

Call data arguments:
item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.
value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.
renderContext - the render context to be used in drawing the item’s background
columnPosition - the Integer position of the column of the cell whose background needs to be drawn.
heading - indicates whether the cell whose background needs to be drawn is a heading cell.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may ‘undo’ the selection change by setting the doit field of the callData to false.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

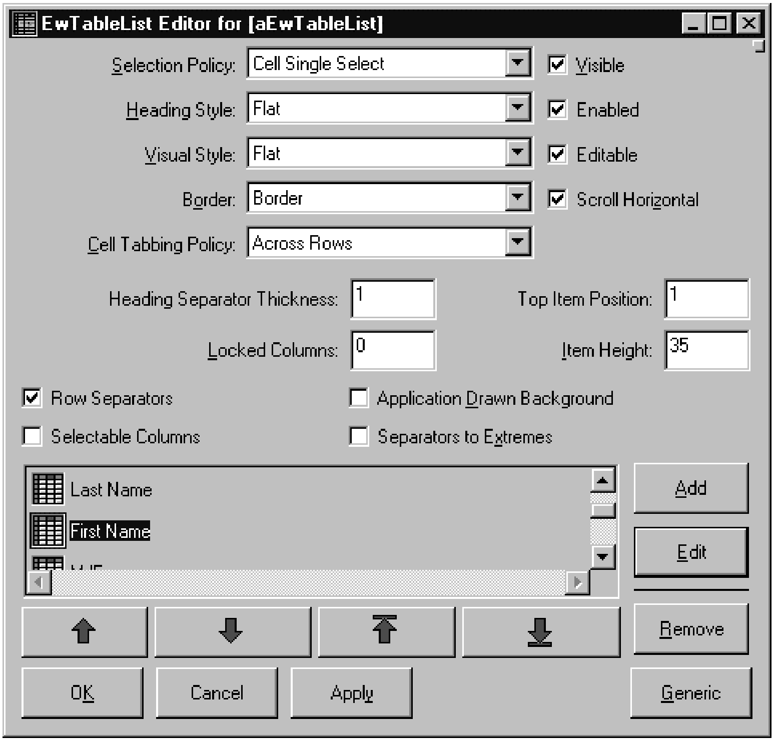
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Visual Info Callback

These callbacks are triggered when an item's icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:
    item - the item which is the selected item.
    itemPosition - the integer position of the selected item in the list.
    isInUse - the inUse status (Boolean) which is the default to be used for the item in
        the callback.

Editor



EwTableList Editor for [aEwTableList]

Selection Policy: Cell Single Select
Heading Style: Flat
Visual Style: Flat
Border: Border
Cell Tabbing Policy: Across Rows

☑ Visible
☑ Enabled
☑ Editable
☑ Scroll Horizontal

Heading Separator Thickness: 1
Top Item Position: 1
Locked Columns: 0
Item Height: 35

☑ Row Separators
☐ Application Drawn Background
☐ Selectable Columns
☐ Separators to Extremes

Last Name
First Name

Application Drawn Background

Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Cell Tabbing Policy

Specifies how tabbing should occur amongst cells.

Across Rows - When the last cell in a row is reached when tabbing forward, go to the first cell in the next row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the previous row.
Application Defined - Tabbing behavior is defined by the application. To use this option, the application must hook the Cell Tab Callback.
Within Row - When the last cell in a row is reached when tabbing forward, go to the first cell in the same row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the same row.

Editable

Specifies whether the cells in the table are editable. For a cell to be editable, the table widget must be editable, the column must be editable, and the application must hook the Begin Edit Callback and set the callData doit to true

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Heading Font

Specifies the font list associated with the headings.

Heading Separator Thickness

Specifies the thickness of the horizontal line separating the column headings from the rest of the table.

Heading Visual Style

Defines the how the cells are displayed.

Etch Cells - Each cell in an etched column is individually etched.
Etch Rows - Each row is etched.
Flat - Looks like a list box.

Item Height

Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Items
An array of objects that are to be displayed as the list items.

Locked Columns
Specifies the number of columns to be locked down on the left side of the table. A column can be locked only if its preceding column is also locked. When scrolling, the locked columns remain fixed on the left side and all other columns scroll under the locked columns. This value must be no greater than the number of columns.

Row Separators
Specifies whether the rows are separated by a horizontal line.

Scroll Horizontal
This resource specifies whether a horizontal scroll bar should be used for the list.

Selectable Columns
Defines whether the user may select columns by clicking on their headings.

Selected Items
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

Selection Policy
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select
Cell Block Select - Allows selection of a rectangular block of cells.
Cell Single Select - Allows single selection of cells.
Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
Read Only Select - Allows navigation, but no selection or callbacks.
Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

Separators To Extremes
Defines whether row and column separators are to be extended to the extreme right and bottom edges of the table.

Top Item Position
Specifies the Integer position of the item that is the first visible item in the list.

Visible

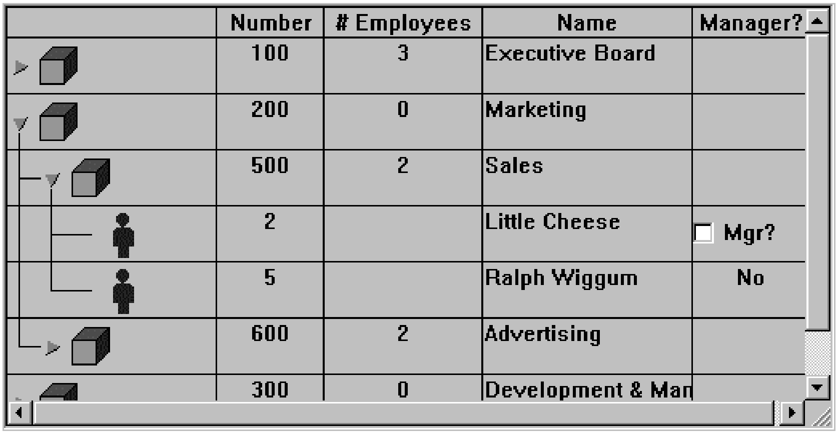
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Visual Style

Defines the how the cells are displayed.

Etch Cells - Each cell in an etched column is individually etched.
Etch Rows - Each row is etched.
Flat - Looks like a list box.

EwTableTree



	Number	# Employees	Name	Manager?
▸	100	3	Executive Board	
▾	200	0	Marketing	
▾	500	2	Sales	
	2		Little Cheese	□ Mgr?
	5		Ralph Wiggum	No
▸	600	2	Advertising	
	300	0	Development & Man	

EwTableTree provides a vertical, multi-column tree of items which shows a row of cells for each item in the list. The items in the list are typically actual application objects, not strings or arrays. For example, the items (rows) might be employee objects, and the columns might show the name, salary, and hire date for each employee. The application should only set the root level items in the tree as the items in the widget. The descendants of those root items can then be shown via `#expandPos:notify:` et al.

The hierarchical aspect of the list is handled by the visualInfoCallback and the childrenCallback. The application must hook the visualInfoCallback and set the callData hasChildren to true or false to indicate whether the item in callData item has children. The application must also hook the childrenCallback and set the callData value to be the list of children for the item in callData item. The childrenCallback only fires for items for which hasChildren is true.

The hierarchyPolicy determines how the hierarchy is to be shown. This includes what the indentation level should be as well as whether to draw lines connecting the items and whether to show some kind of button beside items which have children. By default the hierarchyPolicy is an instance of EwHierarchyPolicy with lines set to true. The other hierarchy policy class provided is EwIconHierarchyPolicy. This class shows an icon beside each item to act as an expand/collapse button. The application can specify which icon to use in different situations so that the button can animate properly as it is pressed. The indentation, lines, and buttons are only shown in the first column.

The application must define the columns in the table by providing an array of EwTableColumn instances. These each define a single column in the table. In addition to defining the heading, width and other visual parameters, the column also provides a cellValueCallback. The application *must* hook this callback on each column. This callback fires whenever the widget needs to know what value to put in a cell for a given item. The application must then set the callData value to be the renderable object (see below) which is to reside in the cell. For example, when the widget needs to place an employee's salary in the salary column, it fires the callValueCallback. The callData tells for which employee the salary is needed (callData item). The application might hook the cellValueCallback from the Salary column to a method like:

```
salary: aWidget clientData: clientData callData: callData
   "The widget needs an employee's salary."
   | employee |
   employee := callData item.
   callData value: employee salary.
```

The application may or may not hook the headingCellValueCallback. Column headings can be set via the EwTableColumn>>heading: method or via the headingCellValueCallback.

Both the headingCellValueCallback and cellValueCallback can be used to set color for a particular cell. Colors can be specified for the foreground, background, selectedForeground, and selectedBackground of a cell

The objects used for cell values and headings are typically Strings, Numbers, and CgIcons, but this is not a requirement. It is possible for an application to use any object as the cell value or heading. Any item which understands `#ewHeightUsing:`, `#ewWidthUsing:` and `#ewDrawUsing:` (called a "renderable object") can be used for a cell value. Since String, Number, and CgIcon already implement the standard renderable object protocol, they can easily be used. Also, since Object provides default renderable behavior, any object can be rendered, although the default rendering is to render the object's printString.

Cell editing can take place automatically or under program control. For cell editing to be automatic, the widget must have editable true, and the columns in which cells may be edited must also have editable true. Under program control, this is not required.

When a cell is about to be edited, either because the user clicked in it or because the program called `#editCellAt:` or `#editSelectedCell`, the column fires its beginEditCallback. The application *must* hook this callback or else editing will never occur. At a minimum, the application must set the callData doit: true to allow editing to begin.

The callData also includes a default editPolicy. The edit policy defines the type of widget to be used for editing as well as some other edit semantics. The default edit policy is an EwTextEditPolicy set up to use a single-line CwText as the edit widget. The application can substitute a more appropriate edit policy for the cell about to be edited. For example, if the cell contains a day of the week, the application may wish to use an EwComboBoxEditPolicy. Applications can define their own custom edit policies by overriding the behavior in EwEditPolicy as required. The supplied subclasses of EwEditPolicy serve as good examples for this.

When editing is about to end, or when the value in the edit widget has been changed (the exact details of when a change has occurred depend on the edit policy), the column fires its endEditCallback. The callData includes the old value and the new value. The application should hook this callback and use the newValue to change some application object. The cell is then automatically refreshed after the callback fires, so the new value is obtained and displayed.

Note: *This widget does not provide scrollbars on is own. In order to have scrollbars, it must be placed within a CwScrolledWindow instance. The widget may be either initially created within the scrolled window, or it may be dragged into an existing scrolled window. Once this has been done, the two widgets are coupled such that they may not be separated.*

Protocol

applicationDrawnBackground: *aBoolean*
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

cellTabbingPolicy: *anInteger*
Specifies how tabbing should occur amongst cells.

Default: XmACROSSROWS (Across Rows)
Valid resource values:

XmACROSSROWS (Across Rows) - When the last cell in a row is reached when tabbing forward, go to the first cell in the next row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the previous row.

XmAPPLICATIONDEFINED (Application Defined) - Tabbing behavior is defined by the application. To use this option, the application must hook the Cell Tab Callback.

XmWITHINROW (Within Row) - When the last cell in a row is reached when tabbing forward, go to the first cell in the same row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the same row.

editable: *aBoolean*
Specifies whether the cells in the table are editable. For a cell to be editable, the table widget must be editable, the column must be editable, and the application must hook the Begin Edit Callback and set the callData doit to true

headingFont: *aString*
Specifies the font list associated with the headings.

headingSeparatorThickness: *anInteger*
Specifies the thickness of the horizontal line separating the column headings from the rest of the table.

headingVisualStyle: *anInteger*
Defines the how the cells are displayed.

Default: XmETCHCELLS (Etch Cells)
Valid resource values:
XmFLAT (Flat) - Looks like a list box.
XmETCHROWS (Etch Rows) - Each row is etched.
XmETCHCELLS (Etch Cells) - Each cell in an etched column is individually etched.

itemHeight: *anInteger*
Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

items: *anOrderedCollection*
An array of objects that are to be displayed as the list items.

lockedColumns: *anInteger*
Specifies the number of columns to be locked down on the left side of the table. A column can be locked only if its preceding column is also locked. When scrolling, the locked columns remain fixed on the left side and all other columns scroll under the locked columns. This value must be no greater than the number of columns.

rowSeparators: *aBoolean*
Specifies whether the rows are separated by a horizontal line.

scrollHorizontal: *aBoolean*
This resource specifies whether a horizontal scroll bar should be used for the list.

selectableColumns: *aBoolean*
Defines whether the user may select columns by clicking on their headings.

selectedItems: *anOrderedCollection*
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:
XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select
XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select
XmREADONLYSELECT (Read Only Select) - Allows navigation, but no selection or callbacks.
XmCELLSINGLESELECT (Cell Single Select) - Allows single selection of cells.
XmCELLBLOCKSELECT (Cell Block Select) - Allows selection of a rectangular block of cells.

separatorsToExtremes: *aBoolean*
Defines whether row and column separators are to be extended to the extreme right and bottom edges of the table.

topItemPosition: *anInteger*
Specifies the Integer position of the item that is the first visible item in the list.

visualStyle: *anInteger*
Defines the how the cells are displayed.

Default: XmFLAT (Flat)
Valid resource values:
XmFLAT (Flat) - Looks like a list box.
XmETCHROWS (Etch Rows) - Each row is etched.
XmETCHCELLS (Etch Cells) - Each cell in an etched column is individually etched.

Callbacks & Events

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Cell Block Selection Callback

These callbacks are triggered when a block of cells is selected in cell block selection mode.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.
columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Cell Single Selection Callback

These callbacks are triggered when an item is selected in cell single selection mode.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.
columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Children Callback

These callbacks are triggered when an item’s list of children is needed.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
children - the value of children for the item in the callback.

Column Heading Selection Callback

These callbacks are triggered when a column heading is selected

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.
columnPosition - the Integer position of the selected column.
columnPositions - a Collection of Integers representing the selected columns.
selectedCells - a Set of Points representing the selected cells.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Draw Background Callback

These callbacks are triggered when an item’s background needs to be drawn.

Call data arguments:

item - the item which is the selected item.
doit - indicates whether the action that invoked the callback is performed.
value - the value which is the renderable whose background needs drawing.
selected - indicates whether the item whose background needs to be drawn is selected.
renderContext - the render context to be used in drawing the item’s background
columnPosition - the Integer position of the column of the cell whose background needs to be drawn.
heading - indicates whether the cell whose background needs to be drawn is a heading cell.

Expand Collapse Callback

These callbacks are triggered when an item is expanded or collapsed.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Modify Verify Callback

These callbacks are triggered when the selection is about to be changed. The application may 'undo' the selection change by setting the doit field of the callData to false.

Call data arguments:

doit - Indicates whether the action that invoked the callback is performed. Setting doit to false negates the action.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of items which are the selected items.

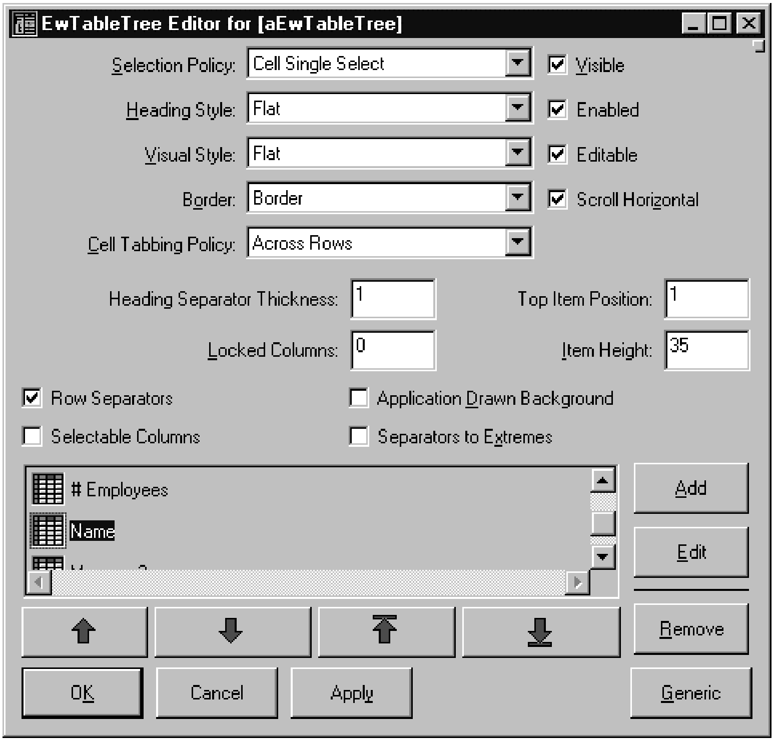
Visual Info Callback

These callbacks are triggered when an item's icon, label and isInUse are needed. The application MUST hook this callback and set the callData icon to the CgIcon (or other renderable object) to be displayed as the icon for the item in callData item. It must also set the callData label to the String (or other renderable object) to be displayed as the label for the item.

Call data arguments:

item - the item which is the selected item.
itemPosition - the integer position of the selected item in the list.
isInUse - the inUse status (Boolean) which is the default to be used for the item in the callback.
icon - the icon which is the default icon to be used for the item in the callback.
label - the label which is the default label to be used for the item in the callback.
hasChildren - indicates whether the item has children or not.

Editor



EwTableTree Editor for [aEwTableTree]

Selection Policy: Cell Single Select
Heading Style: Flat
Visual Style: Flat
Border: Border
Cell Tabbing Policy: Across Rows

☑ Visible
☑ Enabled
☑ Editable
☑ Scroll Horizontal

Heading Separator Thickness: 1
Top Item Position: 1
Locked Columns: 0
Item Height: 35

☑ Row Separators
□ Application Drawn Background
□ Selectable Columns
□ Separators to Extremes

Employees
Name

Application Drawn Background
Specifies whether the application wants the Draw Background Callback to fire when an item's background needs to be drawn. If this value is false, then the widget will draw the normal default background for each selected and non-selected item.

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Cell Tabbing Policy
Specifies how tabbing should occur amongst cells.

Across Rows - When the last cell in a row is reached when tabbing forward, go to the first cell in the next row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the previous row.

Application Defined - Tabbing behavior is defined by the application. To use this option, the application must hook the Cell Tab Callback.

Within Row - When the last cell in a row is reached when tabbing forward, go to the first cell in the same row. When the first cell in a row is reached when tabbing backwards, go to the last cell in the same row.

Editable

Specifies whether the cells in the table are editable. For a cell to be editable, the table widget must be editable, the column must be editable, and the application must hook the Begin Edit Callback and set the callData doit to true

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Heading Font

Specifies the font list associated with the headings.

Heading Separator Thickness

Specifies the thickness of the horizontal line separating the column headings from the rest of the table.

Heading Visual Style

Defines the how the cells are displayed.

Etch Cells - Each cell in an etched column is individually etched.
Etch Rows - Each row is etched.
Flat - Looks like a list box.

Item Height

Specifies the height in pixels of items in the list. This includes the margin height on the top and bottom of the item as well as two pixels for emphasis.

Items

An array of objects that are to be displayed as the list items.

Locked Columns

Specifies the number of columns to be locked down on the left side of the table. A column can be locked only if its preceding column is also locked. When scrolling, the locked columns remain fixed on the left side and all other columns scroll under the locked columns. This value must be no greater than the number of columns.

Row Separators

Specifies whether the rows are separated by a horizontal line.

Scroll Horizontal

This resource specifies whether a horizontal scroll bar should be used for the list.

Selectable Columns
Defines whether the user may select columns by clicking on their headings.

Selected Items
An OrderedCollection of Objects that represents the list items that are currently selected, either by the user or the application.

Selection Policy
Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select
Cell Block Select - Allows selection of a rectangular block of cells.
Cell Single Select - Allows single selection of cells.
Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.
Read Only Select - Allows navigation, but no selection or callbacks.
Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select

Separators To Extremes
Defines whether row and column separators are to be extended to the extreme right and bottom edges of the table.

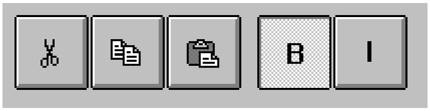
Top Item Position
Specifies the Integer position of the item that is the first visible item in the list.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Visual Style
Defines the how the cells are displayed.

Etch Cells - Each cell in an etched column is individually etched.
Etch Rows - Each row is etched.
Flat - Looks like a list box.

EwToolBar



A tool bar (EwToolbar) provides an interface for building a horizontal or vertical bar containing user interface tools such as push buttons and labels. You can use tool bar widgets to implement the tool bars commonly found under the menu bar in GUI applications. You can use them to provide rows or columns of tools anywhere else in a window. Further, you can use them to implement status bars that allow applications to display various kinds of messages.

Several resources can control the way that a tool bar displays its tools. The *numColumns* resource can specify whether the tools should be arranged in columns. If the numColumns is 0, the tool bar lays out the tools in a row. Setting *numColumns* to 1 produces a vertical tool bar. The spacing resource determines how much space to leave between each tool.

You can specify the colors of the tool bar using the standard *foregroundColor* and *backgroundColor* resources. In addition, you can specify the default colors of tools using the *toolForegroundColor* and *toolBackgroundColor* resources.

The tool bar usually does not have widgets as children. Instead, it has specialized children called tools (EwTool). Tools are user interface elements that look like widgets but do not actually use all of the platform resources required by a widget. Tools collaborate with the parent tool bar to display themselves and handle user events. Tools are lighter weight than widgets and minimize platform resources. However, they do not always look like a platform widget.

Tools on a tool bar can be used much like widgets. Resources such as *width*, *height*, *borderWidth*, *foregroundColor*, and *backgroundColor* can control the appearance of the tool. The *variableWidth* resource controls whether a tool shrinks or grows as the tool bar resizes.

The *enterNotifyCallback* and *leaveNotifyCallback* can determine when the mouse enters or leaves the location on the tool bar occupied by a tool. These callbacks are useful for implementing features such as bubble help or tool tips or for updating a status area.

The most commonly used tools on a tool bar are primitive tools. These are tools that behave much like simple widgets such as buttons and labels. Primitive tools have an image which determines the text or graphics that they display. The image can be any renderable object.

Label tools (EwLabelTool) display an image and provide a shadowed inset as specified in the *shadowType* resource. They provide resources for setting margins similar to the CwLabel widget. Progress bar tools (EwProgressBarTool) look and behave much like EwProgressBar. The button tools (EwPushButtonTool and EwToggleButtonTool) also behave much like their CwWidget counterparts. A separator tool (EwSeparatorTool) can separate tools or clusters of tools. The *separatorType* resource determines the exact appearance of a separator tool.

Groups (EwGroupTool) are useful for grouping multiple tools in order to assign common properties or provide specialized behavior for the group. The *shadowType* resource determines what kind of border is drawn around the tools within a group when the *borderWidth* is not 0.The *spacing* resource determines what spacing the group uses for its tools. If the group contains toggle button tools, the *radioBehavior* resource can enforce radio button style behavior within a group.

Protocol

createGroup: *name* **argBlock:** *argBlock*
Creates a group tool inside the tool bar. The first argument is the name for the new tool; the second is its argument block. A group is a tool that contains other tools.

createLabelTool: *name* **argBlock:** *argBlock*
Creates a label tool inside the tool bar.

createProgressBarTool: *name* **argBlock:** *argBlock*
Creates a progress bar tool inside the tool bar.

createPushButtonTool: *name* **argBlock:** *argBlock*
Creates a push button tool inside the tool bar.

createRadioButtonTool: *name* **argBlock:** *argBlock*
Creates a radio button tool inside the tool bar.

createSeparatorTool: *name* **argBlock:** *argBlock*
Creates a separator tool inside the tool bar.

createToggleButtonTool: *name* **argBlock:** *argBlock*
Creates a toggle button tool inside the tool bar.

drawPolicy: *anInteger*
Specifies the drawing policy used in rendering buttons on the toolbar. The drawing policy determines two things for a button:

- how the button draws itself so it looks like a button
- how the button animates when pressed

Default: XmSHADOWEDTHREESTATEDRAWPOLICY (Shadowed Three State)
Valid resource values:

XmSHADOWEDTWOSTATEDRAWPOLICY (Shadowed Two State) - Buttons are drawn with a 3D shadowed outline and exhibit a simple 2-state (OFF and ON) state rendering.

XmSHADOWEDTHREESTATEDRAWPOLICY (Shadowed Three State) - Buttons are drawn with a 3D shadowed outline and exhibit a 3-state (OFF, ON, and PRESSED) state rendering.

XmOUTLINEDRAWPOLICY (Outlined) - Buttons are drawn with a simple etched (non-shadowed) outline and exhibit a 2-state (OFF and ON) state rendering.

imageHeight: *anInteger*
Specifies the preferred face height of button tools added to the toolbar or to groups. The button face is the area inside the button's beveled edging

imageWidth: *anInteger*
Specifies the preferred face width of button tools added to the toolbar or to groups. The button face is the area inside the button's beveled edging.

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the label, and the top edge of the bottom shadow and the label.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the label, and the left edge of the right shadow and the label.

notifyAlways: *aBoolean*
Specifies whether enter and leave notification callbacks will be generated from notifiable tools on the toolbar anytime the mouse is moved over a notifiable tool, regardless of the state of the mouse buttons. If resourceValue is true, then notification callbacks will be generated as the mouse is moved over the toolbar, even if a button select sequence is pending. If resourceValue is false, then notification callbacks will only be generated if no mouse button is pressed. Once the active mouse button is pressed on a notifiable tool, such as to start a button select sequence, no further notification callbacks will occur until the mouse button is released.

numColumns: *anInteger*
Specifies the number of columns that are made to accommodate the receiver’s children tools. This attribute always sets the x-axis dimension. A value of nil or 0 indicates that tools on the toolbar will be layed out horizontally in one row. A value greater than 0 indicates that tools on the toolbar will be layed out in the specified number of columns and using as many rows as are required. Layout is performed by filling all columns in a row first before creating additional rows. NOTE: This resource applies to the receiver’s children (not the receiver’s descendents). For example, children of the receiver that are group tools will be treated as one tool during layout.

selectiveBorder: *anInteger*
Specifies which edges of the widget are outlined. Edges may be selectively outlined to allow for fine tuning of the visual appearance when the toolbar is placed on a main window.

Default: XmBORDERNONE (None)
Valid resource values:
XmBORDERNONE (None) - No border is drawn on any edge.
XmBORDERLEFTMASK (Left) - A border is drawn on the left edge.
XmBORDERRIGHTMASK (Right) - A border is drawn on the right edge.
XmBORDERTOPMASK (Top) - A border is drawn on the top edge.
XmBORDERBOTTOMMASK (Bottom) - A border is drawn on the bottom edge.
XmBORDERALL (All) - A border is drawn on all edges.

spacing: *anInteger*
Specifies the horizontal and vertical spacing between items contained within the toolbar. The default value is one pixel.

Callbacks & Events

None

EwGroupTool

EwGroupTool is a class whose instances represent groups of tools on a toolbar. Any class of EwTool may be added as a child of the group.

Protocol

drawPolicy: *anInteger*
Specifies the drawing policy used in rendering buttons in this group. If the policy is not set on the group, then inherit the policy from the group’s parent. The drawing policy determines two things for a button:

- how the button draws itself so it looks like a button;
- how the button animates when pressed.

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool's image, and between the top edge of the bottom shadow and the bottom of the tool's image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool's image, and between the left edge of the right shadow and the right edge of the tool's image.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

numColumns: *anInteger*
Specifies the number of columns that are made to accommodate entries for tools in the group. This attribute always sets the x-axis dimension. A value of nil or 0 indicates that tools in the group will be layed out horizontally in one row. A value greater than 0 indicates that tools in the group will be layed out in the specified number columns, using as many rows as are required. Layout is performed by filling all columns in a row first before creating additional rows. NOTE: This resource applies to the receiver's children (not the receiver's descendents'). For example, children of the receiver that are group tools will be treated as one tool during layout.

radioBehavior: *aBoolean*
Specifies a Boolean value that when true, indicates that the group should enforce a RadioBox-type behavior on all button children of the group which are configured as toggle buttons. RadioBox behavior dictates that when one toggle is selected then another toggle is selected, the first toggle is unselected automatically. The default value is false.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

shadowType: *anInteger*
Specifies the drawing style for the frame around the group.

Default: XmSHADOWNONE (None)
Valid resource values:

XmSHADOWNONE (None) - No frame is drawn around the group

XmSHADOWOUTLINE (Outline) - A frame is drawn that appears as an etched line. The borderWidth must be 2 for this to appear etched; if borderWidth is 1, appearance will be as "In".

XmSHADOWIN (In) - A frame is drawn that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed

XmSHADOWOUT (Out) - A frame is drawn that appears outset

spacing: *anInteger*
Specifies the horizontal and vertical spacing between items contained within the toolbar. The default value is one pixel.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

Callbacks & Events

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

Resize Callback
These callbacks are triggered when the widget is resized.

EwLabelTool

EwLabelTool is a class whose instances represent label tools on a toolbar. LabelTools are similar to CwLabel widgets.

Protocol

horizontalAlignment: *anInteger*
Specifies the horizontal alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:

XmALIGNMENTBEGINNING (Left) - Causes the center of the image to be horizontally aligned with the left edge of the tool's window.
XmALIGNMENTCENTER (Center) - Causes the center of the image to be horizontally aligned in the center of the tool's window.
XmALIGNMENTEND (Right) - Causes the center of the image to be horizontally aligned with the right edge of the tool's window.

image: *aRenderableObject*
Specifies the renderable object which draws on the face of the tool.

marginBottom: *anInteger*
Specifies additional spacing, in pixels, that should exist at the below the tool’s label. This resource is used in conjunction with marginHeight.

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool’s image, and between the top edge of the bottom shadow and the bottom of the tool’s image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool’s image, and between the left edge of the right shadow and the right edge of the tool’s image.

marginLeft: *anInteger*
Specifies additional spacing, in pixels, that should exist at the to the left of the tool’s label. This resource is used in conjunction with marginWidth.

marginRight: *anInteger*
Specifies additional spacing, in pixels, that should exist at the to the right of the tool’s label. This resource is used in conjunction with marginWidth.

marginTop: *anInteger*
Specifies additional spacing, in pixels, that should exist at the above the tool’s label. This resource is used in conjunction with marginHeight.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

shadowType: *anInteger*
Specifies the drawing style for the frame around the label.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool

will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

verticalAlignment: *anInteger*
Specifies the vertical alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTCENTER (Center) - Causes the center of the image to be vertically aligned in the center of the tool's window.
XmALIGNMENTTOP (Top) - Causes the top edge of the image to be vertically aligned with the top edge of the tool's window.
XmALIGNMENTBOTTOM (Bottom) - Causes the center of the image to be vertically aligned in the center of the tool's window.

Callbacks & Events

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

Resize Callback
These callbacks are triggered when the widget is resized.

EwProgressBarTool

EwProgressBarTool is class whose instances represent progress indicators on a toolbar. The progress is a colored ribbon which represents a fraction of work completed.

Protocol

direction: *anInteger*
Specifies the direction the progress bar moves in.
Default: XmFORWARD (Forward)
Valid resource values:
XmFORWARD (Forward) - The progress bar moves forward. For horizontal progress bars, this is left-to-right. For vertical progress bars, this is top-to-bottom.
XmREVERSE (Reverse) - The progress bar moves backwards. For horizontal progress bars, this is right-to-left. For vertical progress bars, this is bottom-to-top.

fractionComplete: *anInteger*
Specifies the current amount of progress to show in the progress bar. This resource is a

fraction, denoting a number between 0 and 1. For example, 1/10 specifies 10 % complete. 1 represents 100 % complete.

horizontalAlignment: *anInteger*
Specifies the horizontal alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the center of the image to be horizontally aligned with the left edge of the tool's window.
XmALIGNMENTCENTER (Center) - Causes the center of the image to be horizontally aligned in the center of the tool's window.
XmALIGNMENTEND (Right) - Causes the center of the image to be horizontally aligned with the right edge of the tool's window.

image: *aRenderableObject*
Specifies the renderable object which draws on the face of the tool.

marginBottom: *anInteger*
Specifies additional spacing, in pixels, that should exist at the below the tool's label. This resource is used in conjunction with marginHeight.

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool's image, and between the top edge of the bottom shadow and the bottom of the tool's image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool's image, and between the left edge of the right shadow and the right edge of the tool's image.

marginLeft: *anInteger*
Specifies additional spacing, in pixels, that should exist at the to the left of the tool's label. This resource is used in conjunction with marginWidth.

marginRight: *anInteger*
Specifies additional spacing, in pixels, that should exist at the to the right of the tool's label. This resource is used in conjunction with marginWidth.

marginTop: *anInteger*
Specifies additional spacing, in pixels, that should exist at the above the tool's label. This resource is used in conjunction with marginHeight.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

orientation: *anInteger*
Specifies the orientation of the progress bar.

Default: XmHORIZONTAL (Horizontal)
Valid resource values:
XmVERTICAL (Vertical) - Display the progress bar vertically.
XmHORIZONTAL (Horizontal) - Display the progress bar horizontally.

ribbonImage: *aRenderableObject*
Specifies the renderable object which draws as the completed ribbon of the progress bar. This object is drawn instead of a color strip. Note that when a ribbon image is used, a label (the showPercentage or image resources) will be drawn using the imageColor resource, or if that is nil, the foregroundColor resource.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

shadowType: *anInteger*
Specifies the drawing style for the frame around the label.

showPercentage: *aBoolean*
Specifies the whether a label showing the percentage completed is shown in the progress bar. If true, then the string ‘X %’ is show in the progress bar, where X is the percentage of progress completed. If false, then no percentage label is shown.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

verticalAlignment: *anInteger*
Specifies the vertical alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:

- XmALIGNMENTCENTER (Center) - Causes the center of the image to be vertically aligned in the center of the tool's window.
- XmALIGNMENTTOP (Top) - Causes the top edge of the image to be vertically aligned with the top edge of the tool's window.
- XmALIGNMENTBOTTOM (Bottom) - Causes the center of the image to be vertically aligned in the center of the tool's window.

Callbacks & Events

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

Resize Callback
These callbacks are triggered when the widget is resized.

EwPushButtonTool

EwPushButtonTool is a class whose instances represent push buttons on a toolbar. EwPushButtonTools are similar to CwPushButton widgets.

Protocol

horizontalAlignment: *anInteger*
Specifies the horizontal alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:

- XmALIGNMENTBEGINNING (Left) - Causes the center of the image to be horizontally aligned with the left edge of the tool's window.
- XmALIGNMENTCENTER (Center) - Causes the center of the image to be horizontally aligned in the center of the tool's window.
- XmALIGNMENTEND (Right) - Causes the center of the image to be horizontally aligned with the right edge of the tool's window.

image: *aRenderableObject*
Specifies the renderable object which draws on the face of the tool.

imageHeight: *anInteger*
Specifies the height of the tool's face in pixels, not including the border or bevel pixels. If the imageHeight resource is not set, the imageHeight defaults to the image height established by the parent. The imageWidth and imageHeight resources are alternatives to the width and height resources for specifying a button's dimensions. Setting a button's imageWidth and imageHeight allows the button to be sized to accomodate an explicitly sized icon, and removes the need to factor in the pixels added by the bevelling (which could change if the button's draw policy is changed).

imageWidth: *anInteger*
Specifies the width of the tool's face in pixels,not including the border or bevel pixels. If the imageWidth resource is not set, the imageWidth defaults to the image width established by the parent. The imageWidth and imageHeight resources are alternatives to the width and height resources for specifying a button's dimensions. Setting a button's imageWidth and imageHeight allows the button to be sized to accomodate an explicitly sized icon, and removes the need to factor in the pixels added by the bevelling (which could change if the button's draw policy is changed).

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool's image, and between the top edge of the bottom shadow and the bottom of the tool's image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool's image, and between the left edge of the right shadow and the right edge of the tool's image.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized

large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

verticalAlignment: *anInteger*
Specifies the vertical alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTCENTER (Center) - Causes the center of the image to be vertically aligned in the center of the tool's window.
XmALIGNMENTTOP (Top) - Causes the top edge of the image to be vertically aligned with the top edge of the tool's window.
XmALIGNMENTBOTTOM (Bottom) - Causes the center of the image to be vertically aligned in the center of the tool's window.

Callbacks & Events

Activate Callback
These callbacks are triggered when the tool is activated. A tool is activated when the user presses and releases the active mouse button while the pointer is inside that tool. Activating the button also disarms it.

Arm Callback
These callbacks are triggered when the receiver is armed. The button is armed when the active mouse button is pressed inside the button.

Disarm Callback
These callbacks are triggered when the receiver is disarmed. The button is disarmed when the active mouse button is released, regardless of whether it is released inside the button or not.

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

Resize Callback
These callbacks are triggered when the widget is resized.

EwSeparatorTool

EwSeparatorTool is a class whose instances represent separator tools on a toolbar. SeparatorTools are similar to CwSeparator widgets.

Protocol

autoSize: *aBoolean*
Specifies whether the size of the separator automatically sizes to match the size of the enclosing tool. For a vertically oriented separator, setting autoSize to true will result in the height of the separator being set equal to the height of the enclosing group (or toolbar), if the height resource has not been explicitly set. For a horizontally oriented separator, setting autoSize to true will result in the width of the separator being set equal to the width of the enclosing group (or toolbar), if the width resource has not been explicitly set.

horizontalAlignment: *anInteger*
Specifies the horizontal alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the center of the image to be horizontally aligned with the left edge of the tool's window.
XmALIGNMENTCENTER (Center) - Causes the center of the image to be horizontally aligned in the center of the tool's window.
XmALIGNMENTEND (Right) - Causes the center of the image to be horizontally aligned with the right edge of the tool's window.

image: *aRenderableObject*
Specifies the renderable object which draws on the face of the tool.

margin: *anInteger*
For horizontal orientation, specifies the space on the left and right sides between the border of the tool and the drawn line(s). For vertical orientation, specifies the space on the top and bottom between the border of the tool and the drawn line(s)..

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool's image, and between the top edge of the bottom shadow and the bottom of the tool's image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool's image, and between the left edge of the right shadow and the right edge of the tool's image.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

orientation: *anInteger*
Specifies the orientation of the separator.

Default: XmVERTICAL (Vertical)
Valid resource values:
XmVERTICAL (Vertical) - Displays Separator vertically.
XmHORIZONTAL (Horizontal) - Displays Separator horizontally.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

separatorType: *anInteger*
Specifies the type of line drawing to be done in the Separator tool.

Default: XmNOLINE (No Line)
Valid resource values:
XmNOLINE (No Line) - No line.
XmSINGLELINE (Single Line) - Draws Separator using a single line.
XmDOUBLELINE (Double Line) - Draws Separator using a double line.
XmSINGLEDASHEDLINE (Single Dashed Line) - Draws Separator using a single dashed line.
XmDOUBLEDASHEDLINE (Double Dashed Line) - Draws Separator using a double dashed line.
XmSHADOWETCHEDIN (Etched In) - Draws Separator using a double line giving the effect of a line etched into the window.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

verticalAlignment: *anInteger*
Specifies the vertical alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTCENTER (Center) - Causes the center of the image to be vertically aligned in the center of the tool's window.

XmALIGNMENTTOP (Top) - Causes the top edge of the image to be vertically aligned with the top edge of the tool's window.
XmALIGNMENTBOTTOM (Bottom) - Causes the center of the image to be vertically aligned in the center of the tool's window.

Callbacks & Events

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

Resize Callback
These callbacks are triggered when the widget is resized.

EwToggleButtonTool

EwToggleButtonTool is a class whose instances represent toggle button tools on a toolbar. EwToggleButtonTools are similar to CwToggleButton widgets.

Protocol

horizontalAlignment: *anInteger*
Specifies the horizontal alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the center of the image to be horizontally aligned with the left edge of the tool's window.
XmALIGNMENTCENTER (Center) - Causes the center of the image to be horizontally aligned in the center of the tool's window.
XmALIGNMENTEND (Right) - Causes the center of the image to be horizontally aligned with the right edge of the tool's window.

image: *aRenderableObject*
Specifies the renderable object which draws on the face of the tool.

imageHeight: *anInteger*
Specifies the height of the tool's face in pixels, not including the border or bevel pixels. If the imageHeight resource is not set, the imageHeight defaults to the image height established by the parent. The imageWidth and imageHeight resources are alternatives to the width and height resources for specifying a button's dimensions. Setting a button's imageWidth and imageHeight allows the button to be sized to accomodate an explicitly

sized icon, and removes the need to factor in the pixels added by the bevelling (which could change if the button's draw policy is changed).

imageWidth: *anInteger*
Specifies the width of the tool's face in pixels,not including the border or bevel pixels. If the imageWidth resource is not set, the imageWidth defaults to the image width established by the parent. The imageWidth and imageHeight resources are alternatives to the width and height resources for specifying a button's dimensions. Setting a button's imageWidth and imageHeight allows the button to be sized to accomodate an explicitly sized icon, and removes the need to factor in the pixels added by the bevelling (which could change if the button's draw policy is changed).

marginHeight: *anInteger*
Specifies the amount of blank space between the bottom edge of the top shadow and the top of the tool's image, and between the top edge of the bottom shadow and the bottom of the tool's image.

marginWidth: *anInteger*
Specifies the amount of blank space between the right edge of the left shadow and the left edge of the tool's image, and between the left edge of the right shadow and the right edge of the tool's image.

minimumWidth: *anInteger*
Specifies the minimum width of the tool in pixels, not including the border area. This resource is only used when the #variableWidth resource is set to true.

radioBehavior: *aBoolean*
Specifies a Boolean value that indicates whether the tool should collaborate with its enclosing group (or toolbar) and enforce a RadioBox-type behavior between it and all like buttons in the group. RadioBox behavior dictates that when one toggle is selected and another toggle is selected, the first toggle is unselected automatically. The default value is false.

sensitive: *aBoolean*
Determines whether a tool will react to input events. Disabled (insensitive) tools do not react to input events.

set: *aBoolean*
Displays the button in its selected state if set to true.

variableWidth: *aBoolean*
Specifies whether the tool is variable-width. The width of a variable-width tool changes as the width of the parent of the tool changes. If the parent of the tool grows larger, the parent will divide up the available variable width (that not consumed by fixed-width tools) proportionally amongst the variable-width tools. Likewise, if the parent of the tool shrinks smaller, the available variable width is divided up, however the width of a tool

will never be resized to less than its defined minimumWidth (0 by default). The relative size ratios used to divide up the available variable width is determined based on the create-time sizes of the tool and its siblings in the tool hierarchy. If the toolbar is sized large enough that no tool is set to its minimum width, then the widths of the variable-width tools will be maintained proportional to these ratios.

verticalAlignment: *anInteger*
Specifies the vertical alignment for the tool's image.

Default: XmALIGNMENTCENTER (Center)
Valid resource values:

XmALIGNMENTCENTER (Center) - Causes the center of the image to be vertically aligned in the center of the tool's window.
XmALIGNMENTTOP (Top) - Causes the top edge of the image to be vertically aligned with the top edge of the tool's window.
XmALIGNMENTBOTTOM (Bottom) - Causes the center of the image to be vertically aligned in the center of the tool's window.

Callbacks & Events

Activate Callback
These callbacks are triggered when the tool is activated. A tool is activated when the user presses and releases the active mouse button while the pointer is inside that tool. Activating the button also disarms it.

Arm Callback
These callbacks are triggered when the receiver is armed. The button is armed when the active mouse button is pressed inside the button.

Disarm Callback
These callbacks are triggered when the receiver is disarmed. The button is disarmed when the active mouse button is released, regardless of whether it is released inside the button or not.

Enter Notify Callback
These callbacks are triggered when the mouse enters the region occupied by the receiver.

Leave Notify Callback
These callbacks are triggered when the tool is destroyed.

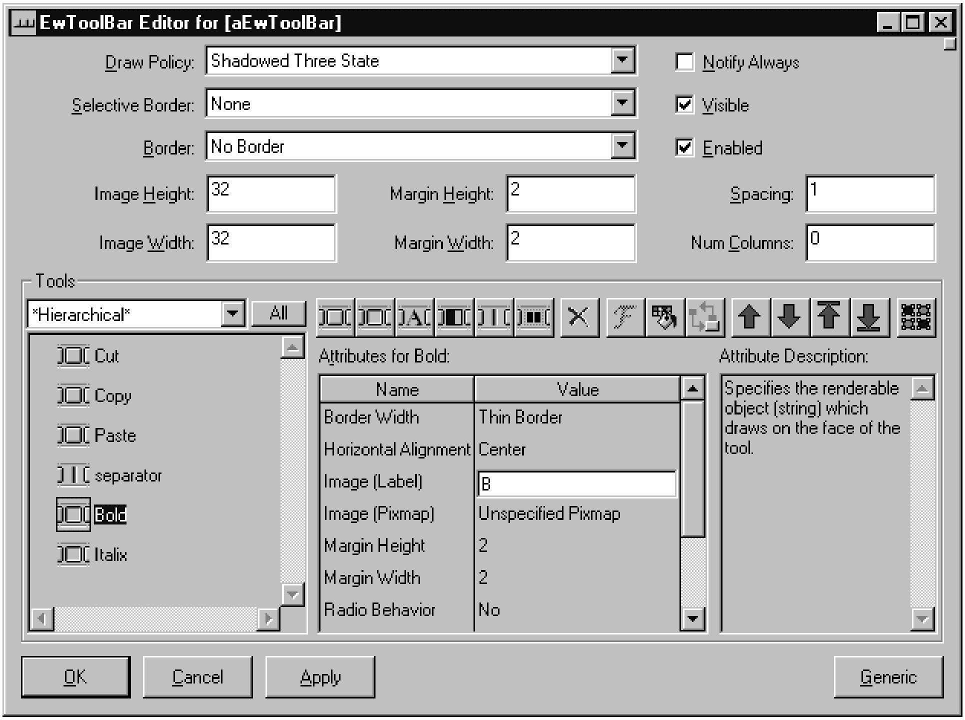
Resize Callback
These callbacks are triggered when the widget is resized.

Value Changed Callback
These callbacks are triggered when the ToggleButton value is changed.

Call data arguments:

set - a Boolean value indicating if the EwToggleButtonTool is toggle on (true) or off (false).

Editor



EwToolBar Editor for [aEwToolBar]

Draw Policy: Shadowed Three State

Selective Border: None

Border: No Border

□ Notify Always

☑ Visible

☑ Enabled

Image Height: 32

Image Width: 32

Margin Height: 2

Margin Width: 2

Spacing: 1

Num Columns: 0

Tools

Hierarchical

Cut

Copy

Paste

separator

Bold

Italix

Attributes for Bold:

Name	Value
Border Width	Thin Border
Horizontal Alignment	Center
Image (Label)	B
Image (Pixmap)	Unspecified Pixmap
Margin Height	2
Margin Width	2
Radio Behavior	No

Attribute Description:

Specifies the renderable object (string) which draws on the face of the tool.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Draw Policy

Specifies the drawing policy used in rendering buttons on the toolbar. The drawing policy determines two things for a button:

- how the button draws itself so it looks like a button
- how the button animates when pressed

Shadowed Two State - Buttons are drawn with a 3D shadowed outline and exhibit a simple 2-state (OFF and ON) state rendering.

Shadowed Three State - Buttons are drawn with a 3D shadowed outline and exhibit a 3-state (OFF, ON, and PRESSED) state rendering.

Outlined - Buttons are drawn with a simple etched (non-shadowed) outline and exhibit a 2-state (OFF and ON) state rendering.

Image Height

Specifies the preferred face height of button tools added to the toolbar or to groups. The button face is the area inside the button’s beveled edging

Image Width

Specifies the preferred face width of button tools added to the toolbar or to groups. The button face is the area inside the button’s beveled edging.

Margin Height

Specifies the amount of blank space between the bottom edge of the top shadow and the label, and the top edge of the bottom shadow and the label.

Margin Width

Specifies the amount of blank space between the right edge of the left shadow and the label, and the left edge of the right shadow and the label.

Notify Always

Specifies whether enter and leave notification callbacks will be generated from notifiable tools on the toolbar anytime the mouse is moved over a notifiable tool, regardless of the state of the mouse buttons. If resourceValue is true, then notification callbacks will be generated as the mouse is moved over the toolbar, even if a button select sequence is pending. If resourceValue is false, then notification callbacks will only be generated if no mouse button is pressed. Once the active mouse button is pressed on a notifiable tool, such as to start a button select sequence, no further notification callbacks will occur until the mouse button is released.

Num Columns

Specifies the number of columns that are made to accommodate the receiver’s children tools. This attribute always sets the x-axis dimension. A value of nil or 0 indicates that tools on the toolbar will be layed out horizontally in one row. A value greater than 0 indicates that tools on the toolbar will be layed out in the specified number of columns and using as many rows as are required. Layout is performed by filling all columns in a row first before creating additional rows. NOTE: This resource applies to the receiver's children (not the receiver's descendents). For example, children of the receiver that are group tools will be treated as one tool during layout.

Selective Border

Specifies which edges of the widget are outlined. Edges may be selectively outlined to allow for fine tuning of the visual appearance when the toolbar is placed on a main window.

None - No border is drawn on any edge.
Left - A border is drawn on the left edge.
Right - A border is drawn on the right edge.
Top - A border is drawn on the top edge.
Bottom - A border is drawn on the bottom edge.
All - A border is drawn on all edges.

Spacing

Specifies the horizontal and vertical spacing between items contained within the toolbar. The default value is one pixel.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Tools

An array of EwTools that are to be displayed within the toolbar.

EwPushButtonTool

EwPushButtonTool is a class whose instances represent push buttons on a toolbar. EwPushButtonTools are similar to CwPushButton widgets.

EwToggleButtonTool

EwToggleButtonTool is a class whose instances represent toggle button tools on a toolbar. EwToggleButtonTools are similar to CwToggleButton widgets.

EwLabelTool

EwLabelTool is a class whose instances represent label tools on a toolbar. LabelTools are similar to CwLabel widgets.

EwProgressBarTool

EwProgressBarTool is class whose instances represent progress indicators on a toolbar. The progress is a colored ribbon which represents a fraction of work completed.

EwSeparatorTool

EwSeparatorTool is a class whose instances represent separator tools on a toolbar. SeparatorTools are similar to CwSeparator widgets.

EwGroupTool

EwGroupTool is a class whose instances represent groups of tools on a toolbar. Any class of EwTool may be added as a child of the group.

Delete

Delete the selected tools from the toolbar.

Font

Set the font for the selected tools.

Color

Set the color for the selected tools.

Callbacks

Set up callbacks for the selected tools.

Up & Down

Move the selected tools up or down in the list.

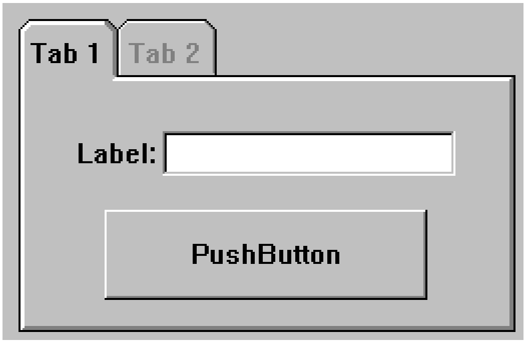
Top & Bottom

Move the selected tools to the top or bottom of the list.

Select All

Select all of the tools in the toolbar.

EwWINNotebook



EwWINNotebook implements a Windows flavor notebook. All of the page's (children) tabs are displayed at the top of the notebook. A tab's height is determined by the tabHeight resource. A tab's width is the page width divided by the number of tabs per row. The top page which is displaying its widgets is represented by the currentPage resource.

Protocol

tabHeight: *anInteger*
Specifies the height of the notebook's tabs in pixels.

tabsPerRow: *anInteger*
Specifies how many page's tabs are displayed in a row before another row is created.

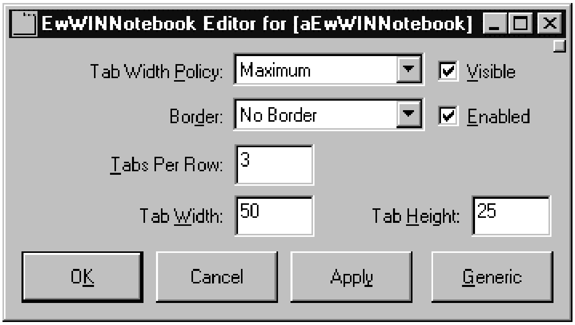
tabWidthPolicy: *anInteger*
Specifies the technique that will be used to set the width of the tabs in a notebook.

Default: XmMAXIMUM (Maximum)
Valid resource values:
XmCONSTANT (Constant) - The tabs will be sized according to the value of the majorTabWidth and minorTabWidth resources.
XmVARIABLE (Variable) - The tabs will be individually sized to fit their labels.
XmMAXIMUM (Maximum) - The tabs will all be the size of the tab needed to accomodate the widest tab label.

Callbacks & Events

Page Change Callback
These callbacks are triggered just before any switching of pages take place.

Editor



Tab Width Policy: Maximum

☑ Visible

Border: No Border

☑ Enabled

Tabs Per Row: 3

Tab Width: 50

Tab Height: 25

Border Width
Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Tab Height
Specifies the height of the notebook’s tabs in pixels.

Tab Per Row
Specifies the number of tabs in each row. When that number us exceeded, a new row is added.

Tab Width
Specifies the width of the notebook’s tabs in pixels.

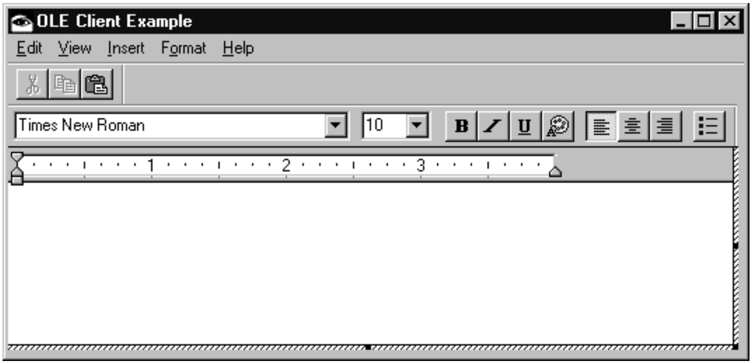
Tab Width Policy
Specifies the technique that will be used to set the width of the tabs in a notebook.

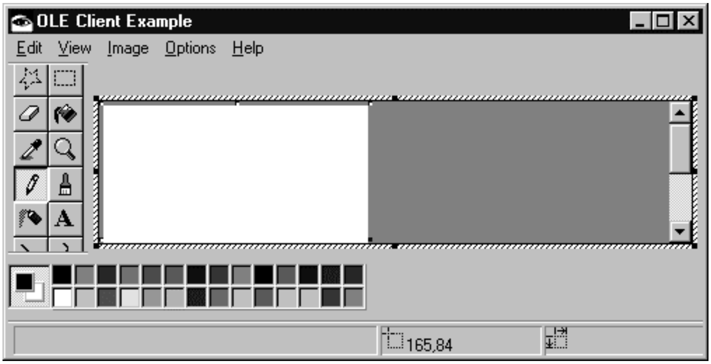
Constant - The tabs will be sized according to the value of the majorTabWidth and minorTabWidth resources.
Variable - The tabs will be individually sized to fit their labels.
Maximum - The tabs will all be the size of the tab needed to accomodate the widest tab label.

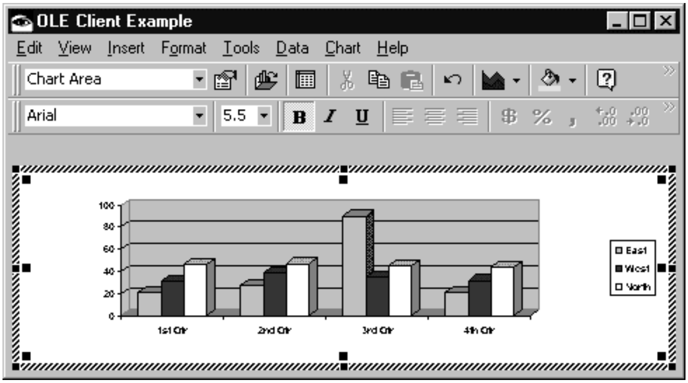
Tabs Per Row
Specifies how many page’s tabs are displayed in a row before another row is created.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

OleClient







OleClient is designed to access services from an OLE server application. It can be used to embed or link OLE objects such as spreadsheets, word processor docs, charts, etc. This widget is only available in the Windows version of VisualAge.

Protocol

activateVerb: *anInteger*
Specifies the operation to be performed on the contained OLE object when it is activated. Activation is controlled by the activationPolicy resource. The activation verb is supplied from the verbs list for the OLE object contained in the receiver.

Default: XmVERBPRIMARY (Primary)
Valid resource values:
XmVERBPROPERTIES (Properties) - Request the OLE object properties dialog
XmVERBDISCARDUNDOSTATE (Discard Undo State) - Close the OLE object and discard the undo state
XmVERBINPLACEACTIVATE (In Place Activate) - Open the OLE for editing in-place
XmVERBUIACTIVATE (UI Activate) - Activate the UI for the OLE object
XmVERBHIDE (Show) - Show the OLE object
XmVERBOPEN (Open) - Open the OLE object for editing in a separate window
XmVERBSHOW (Show) - Show the OLE object
XmVERBPRIMARY (Primary) - Opens the OLE object for editing

activationPolicy: *anInteger*
Specifies the policy used for activation of the OLE object contained by the receiver.

Default: XmACTIVATEDOUBLECLICK (Double Click)
Valid resource values:
XmACTIVATEDOUBLECLICK (Double Click) - The OLE object is activated whenever a double-click is detected on the receiver
XmACTIVATEMANUAL (Manual) - The OLE object can only be activated programatically via the #doVerb: method

clientClipping: *aBoolean*
The client clipping resource is used to control the drawing of embedded OLE objects. When this value is true, the OLE object cannot draw outside of the receiver.

clientName: *aString*
The client name resource is used when the receiver is created to select the OLE class for the OLE object contained in the receiver. The OLE class name is of the form application.objecttype.version or objecttype.version where:

- application - The name of the application that supplies the object.
- objecttype - The object’s name as defined in the registration database.
- version - The version number of the application that supplies the object.

clientType: *anInteger*
The client type resource is used when the receiver is created to select the type of OLE client object that will be contained in the receiver is.

Default: XmEMBEDDED (Embedded)
Valid resource values:
XmUNDEFINED (Undefined) - There is no OLE object contained in the receiver
XmLINKED (Linked) - The OLE object in the receiver is linked
XmEMBEDDED (Embedded) - The OLE object in the receiver is embedded

deactivationPolicy: *anInteger*
Specifies the policy used for deactivation of the OLE object contained by the receiver.

Default: XmDEACTIVATEONLOSEFOCUS (On Lose Focus)
Valid resource values:
XmDEACTIVATEONLOSEFOCUS (On Lose Focus) - The OLE object is deactivated whenever focus is given to another widget in the receiver's shell
XmDEACTIVATEMANUAL (Manual) - The OLE object can only be deactivated programatically via the #doVerb: method

decorationPolicy: *anInteger*
Specifies the decoration policy used for the receiver.

Default: XmBORDER (Border)
Valid resource values:
XmNONE (None) - No special trimmings are displayed around the receiver
XmBORDER (Border) - A border is displayed around the receiver
XmNIBS (Nibs) - Resize nibs are displayed around the receiver
XmBORDERANDNIBS (Border and Nibs) - Border and resize nibs are displayed

displayAsIcon: *aBoolean*
If this value is true, the OLE object is displayed as an icon in the receiver when created. Otherwise, the OLE object is displayed in the default manner.

focusDecorationPolicy: *anInteger*
Specifies the decoration policy used when the receiver does has focus.

Default: XmBORDERANDNIBS (Border and Nibs)
Valid resource values:
XmNONE (None) - No special trimmings are displayed around the receiver
XmBORDER (Border) - A border is displayed around the receiver
XmNIBS (Nibs) - Resize nibs are displayed around the receiver
XmBORDERANDNIBS (Border and Nibs) - Border and resize nibs are displayed

lcid: *anInteger*
Advanced feature: this resource should only be set if the receiver needs its automation objects to use an LCIDother than LocaleSystemDefault.

sizePolicy: *anInteger*
Specifies the policy used for controlling the display bounds of the OLE object contained by the receiver

Default: XmSIZEACTUAL (Actual)
Valid resource values:

XmSIZEACTUAL (Actual) - The OLE object's image is displayed in actual size within the widget. No changes are made to the receiver's or the OLE object's extents

XmSIZESTRETCH (Stretch) - The display of the OLE object's image is stretched to cover the widget's extents. No changes are made to the receiver's or the OLE object's extents

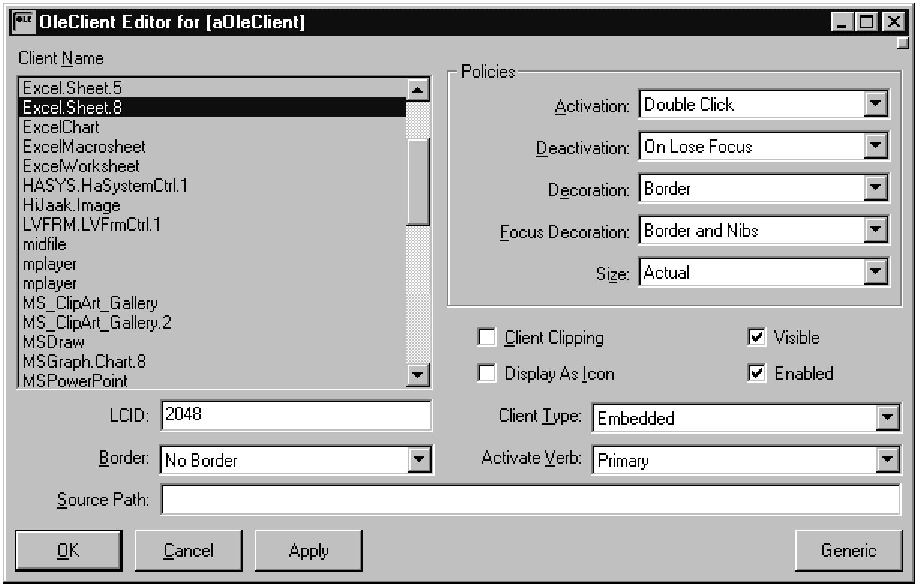
sourcePath: *aString*
When the receiver is created with client type XmEMBEDDED or XmLINKED, this resource is used to specify the source file name for the embedded or linked object.

Callbacks & Events

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



OleClient Editor for [aOleClient]

Client Name

- Excel.Sheet.5
- Excel.Sheet.8
- ExcelChart
- ExcelMacrosheet
- ExcelWorksheet
- HASYS.HaSystemCtrl.1
- HiJaak.Image
- LVFRM.LVFrmCtrl.1
- midfile
- mplayer
- mplayer
- MS_ClipArt_Gallery
- MS_ClipArt_Gallery.2
- MSDraw
- MSGraph.Chart.8
- MSPowerPoint

Policies

Activation: Double Click
Deactivation: On Lose Focus
Decoration: Border
Focus Decoration: Border and Nibs
Size: Actual

☐ Client Clipping ☑ Visible
☐ Display As Icon ☑ Enabled

LCID: 2048
Border: No Border
Client Type: Embedded
Activate Verb: Primary
Source Path:

Activate Verb

Specifies the operation to be performed on the contained OLE object when it is activated. Activation is controlled by the activationPolicy resource. The activation verb is supplied from the verbs list for the OLE object contained in the receiver.

Properties - Request the OLE object properties dialog
Discard Undo State - Close the OLE object and discard the undo state
In Place Activate - Open the OLE for editing in-place
UI Activate - Activate the UI for the OLE object
Show - Show the OLE object
Open - Open the OLE object for editing in a separate window
Show - Show the OLE object
Primary - Opens the OLE object for editing

Activation Policy
Specifies the policy used for activation of the OLE object contained by the receiver.

Double Click - The OLE object is activated whenever a double-click is detected
Manual - The OLE object can only be activated programatically via #doVerb:

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Client Clipping
The client clipping resource is used to control the drawing of embedded OLE objects. When this value is true, the OLE object cannot draw outside of the receiver.

Client Name
The client name resource is used when the receiver is created to select the OLE class for the OLE object contained in the receiver. The OLE class name is of the form application.objecttype.version or objecttype.version where:

- application - The name of the application that supplies the object.
- objecttype - The object's name as defined in the registration database.
- version - The version number of the application that supplies the object.

Client Type
The client type resource is used when the receiver is created to select the type of OLE client object that will be contained in the receiver is.

Undefined - There is no OLE object contained in the receiver
Linked - The OLE object in the receiver is linked
Embedded - The OLE object in the receiver is embedded

Deactivation Policy
Specifies the policy used for deactivation of the OLE object contained by the receiver.

On Lose Focus - The OLE object is deactivated whenever focus is given to another widget in the receiver's shell
Manual - The OLE object can only be deactivated programatically via #doVerb:

Decoration Policy
Specifies the decoration policy used for the receiver.

None - No special trimmings are displayed around the receiver
Border - A border is displayed around the receiver
Nibs - Resize nibs are displayed around the receiver
Border and Nibs - Border and resize nibs are displayed

Display As Icon
If this value is true, the OLE object is displayed as an icon in the receiver when created. Otherwise, the OLE object is displayed in the default manner.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Focus Decoration Policy
Specifies the decoration policy used when the receiver does has focus.

None - No special trimmings are displayed around the receiver
Border - A border is displayed around the receiver
Nibs - Resize nibs are displayed around the receiver
Border and Nibs - Border and resize nibs are displayed

LCID
Advanced feature: this resource should only be set if the receiver needs its automation objects to use an LCID other than LocaleSystemDefault.

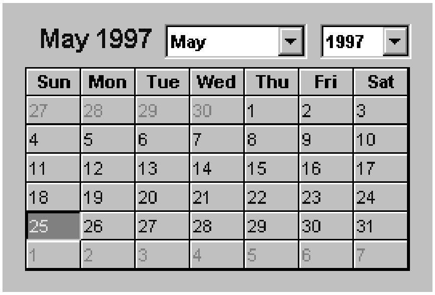
Size Policy
Specifies the policy used for controlling the display bounds of the OLE object contained by the receiver

Actual - The OLE object's image is displayed in actual size within the widget. No changes are made to the receiver's or the OLE object's extents
Stretch - The display of the OLE object's image is stretched to cover the widget's extents. No changes are made to the receiver's or the OLE object's extents

Source Path
When the receiver is created with client type XmEMBEDDED or XmLINKED, this resource is used to specify the source file name for the embedded or linked object.

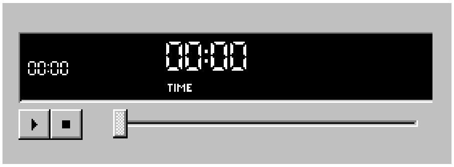
Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

OleControl



May 1997 May 1997

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7



OleControl is used to host OLE/ActiveX controls. This widget is only available in the Windows version of VisualAge.

WindowBuilder Pro provides two code generation mechanisms for generating OleControl properties. Which mechanism is used is controlled via the *Generate OLE Properties* code generation property (settable via the Property Editor). If this property is on (the default), each property setting will be generated as “propertyAt: <*Name*> put: <*value*>”. Settable properties are limited to String, Integer, Float and Boolean. The code in `#addWidgets` will look like this:

```
aOleControl
   propertyAt: ‘BorderStyle’ put: 1;
   propertyAt: ‘Enabled’ put: true;
   propertyAt: ‘FontBold’ put: false;
   propertyAt: ‘FontItalic’ put: false;
   propertyAt: ‘FontName’ put: ‘MS Sans Serif’;
```

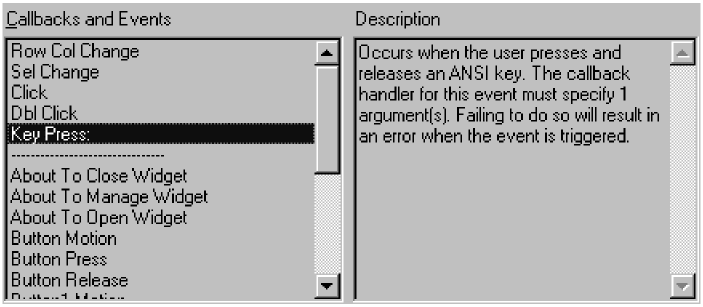
If this property is turned off, the property settings will be generated to an OLE file named <*widgetName*>.CON. All properties are settable, but the .CON file must be present in the working directory. The generated code looks like this:

```
aOleControl
   ifNotNil: [:oleWidget |
      (OleFile open: ‘aOleControl.con’) ifNotNil: [:oleFile |
         oleWidget
            doVerb: -3; “XmVERBHIDE”
            readFromFile: oleFile named: ‘aOleControl’;
            doVerb: -1. “XmVERBSHOW”
         oleFile close]];
```

In the above example, the *aOleControl.con* file *must* be present in the working directory for the widget to pick up its properties when run.

Thus there is a trade off between convenience and expressiveness. Having the properties generated within the `#addWidgets` method is more convenient than having to drag around multiple .CON files. However, this is at the expense of not being able to record all of the property changes. If it is critical that all of the properties settable via the controls property dialog be recorded, then you must use the file-based approach.

When defining behavior for an OLE/ActiveX control, the Callback Editor will list the OLE events generated by the control. These events are somewhat different than normal VisualAge-style events. With OLE events you are strictly limited in the number of arguments allowed in an callback handler. Each event will dictate whether its handlers should have zero or one argument. Zero argument events will not have a colon in their name. One argument events will have a colon. The description of the event will explicitly describe its expectations.



Callbacks and Events

Row Col Change
Sel Change
Click
Dbl Click
Key Press:

About To Close Widget
About To Manage Widget
About To Open Widget
Button Motion
Button Press
Button Release

Description

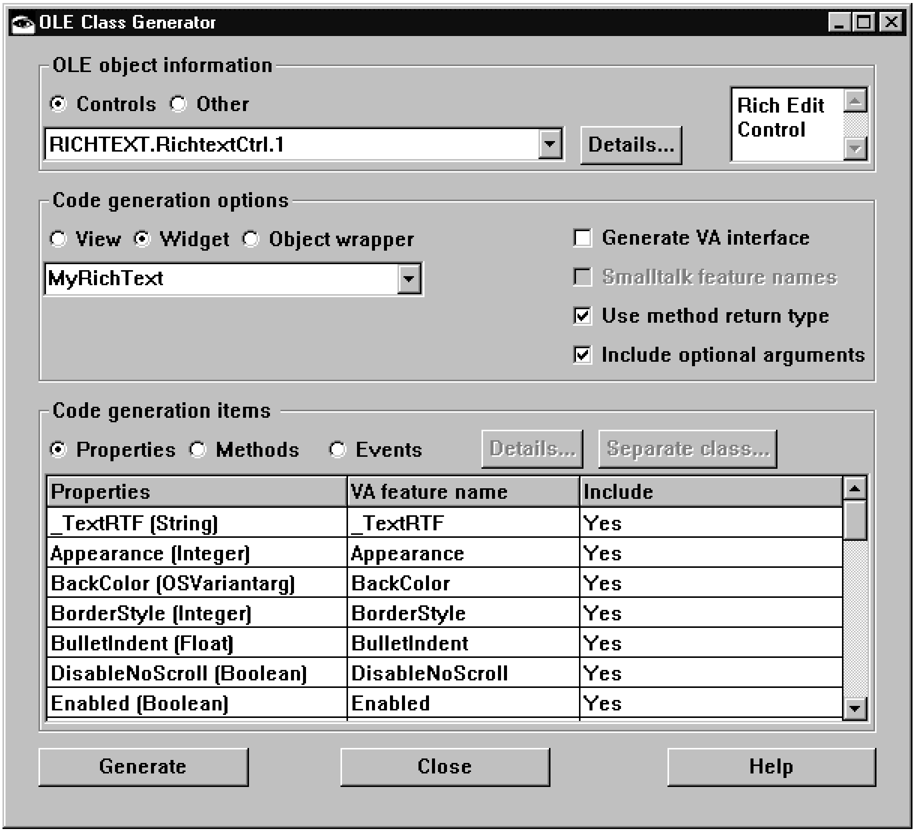
Occurs when the user presses and releases an ANSI key. The callback handler for this event must specify 1 argument(s). Failing to do so will result in an error when the event is triggered.

When specifying the callback selector, make sure that the number of arguments matches that of the event. Failing to do so will result in an argument number mismatch error when the code is executed.

Wrappered OLE/ActiveX Controls

In addition to instances of OleControl, WindowBuilder Pro can also use instances of wrappered OLE/ActiveX controls (subclasses of AbtOleExtendedWidget). These can be created using the procedure described in the section on *Wrapping OLE Controls* in the *VisualAge for Smalltalk User's Guide*. Note that under **Code Generation Options**, you should select **Widget** rather than **View** (**View** creates a part suitable for use with the VisualAge Composition Editor. **Widget** creates a CwWidget/AbtOleExtendedWidget subclass). The OLE Class Generator dialog is shown on the next page with settings appropriate to creating AbtOleExtendedWidget subclasses.

A wrappered OLE/ActiveX widget can be added to the design surface via the **OLE/ActiveX** option on the Add menu. The attribute editor for the widget is the OLE property dialog specific to that control. Note that some properties settable via the property dialog are not code generatable. Code generatable attributes are limited to String, Integer, Float and Boolean. If you need access to all available attributes, use an instance of a generic OleControl and the file-based code generation option.



OLE object information

● Controls ○ Other

RICHTEXT.RichtextCtrl.1

Rich Edit Control

Code generation options

○ View ● Widget ○ Object wrapper

MyRichText

□ Generate VA interface

□ Smalltalk feature names

☑ Use method return type

☑ Include optional arguments

Code generation items

● Properties ○ Methods ○ Events

Properties	VA feature name	Include
_TextRTF (String)	_TextRTF	Yes
Appearance (Integer)	Appearance	Yes
BackColor (OSVariantarg)	BackColor	Yes
BorderStyle (Integer)	BorderStyle	Yes
BulletIndent (Float)	BulletIndent	Yes
DisableNoScroll (Boolean)	DisableNoScroll	Yes
Enabled (Boolean)	Enabled	Yes

Protocol

addOleEventHandler: *eventName* **receiver:** *object* **selector:** *selector*
Register a OLE event handler for the receiver that will cause the selector to be sent to the object when the event is triggered. Answer true if the event handler was successfully registered.

- eventName - a String or DISPID. A DISPID is a well known Integer value. Each OLE event is identified by name or by DISPID.

ambientPropertyReceiver: *anObject*
Specifies the object to be queried (by sending the message specified by the *ambientPropertySelector* resource) to answer ambient property values for the receiver. If the XmNambientPropertyReceiver resource is nil, there is no ambient property support. This resource cannot be changed after the receiver is created..

ambientPropertySelector: *aSymbol*
Specifies the message to be sent (to the value of the *ambientPropertyReceiver* resource) to answer ambient property values for the receiver.

clientName: *aString*
The client name resource is used when the receiver is created to select the OLE class for the OLE object contained in the receiver. The OLE class name is of the form application.objecttype.version or objecttype.version where:

- application - The name of the application that supplies the object.
- objecttype - The object's name as defined in the registration database.
- version - The version number of the application that supplies the object.

invoke: *methodName* **withArguments:** *argumentsArray*
Invoke the named OLE method with arguments for the OLE automation server. If the method name is not found, answer nil. Otherwise, the return value is the result of executing the named OLE method.

- methodName - a String or DISPID. A DISPID is a well known Integer value. Each OLE method is identified by name or by DISPID.
- argumentsArray - an Array of Smalltalk objects. Simple Smalltalk objects are automatically converted into their OLE equivalents based on their class. The following conversions are supported: Integer, Float, String, true, false, OleAutomationObject. A value of nil is used to specify an OLE optional argument.

invoke: *methodName* **withArguments:** *argumentsArray* **returnType:** *returnType*
Invoke the named OLE method with arguments for the OLE automation server. If the method name is not found, answer nil. Otherwise, the return value is derived from the returnType as follows:

    - returnType = false : ignore return value
    - returnType = true : answer return value as returned by the OLE method
    - returnType = valid VARTYPE : answer return value coerced to returnType

- methodName - a String or DISPID. A DISPID is a well known Integer value. Each OLE method is identified by name or by DISPID.
- argumentsArray - an Array of Smalltalk objects. Simple Smalltalk objects are automatically converted into their OLE equivalents based on their class. The following conversions are supported: Integer, Float, String, true, false, OleAutomationObject. A value of nil is used to specify an OLE optional argument.

lcid: *anInteger*
Advanced feature: this resource should only be set if the receiver needs its automation objects to use an LCID other than LocaleSystemDefault.

licenseKey: *aString*
Specifies the licensing key to use when creating an OLE control. If the license key is nil, then OLE control will be created without attempting to license it. This will result in failure if licensing is required.

propertyAt: *aString*
Return the value of the named property of the OLE object. If the property is not found, this action returns nil. Otherwise, the result is the value returned by the OLE automation server. The property name must be specified as a string or a DISPID. A DISPID is a well known integer value; each OLE property is identified by name or by DISPID.

propertyAt: *aString* **put:** *value*
Set the value of the named property of the OLE object. If the property is not found, this action returns nil. Otherwise, the result is the value returned by the OLE automation server. The property name must be specified as a string or a DISPID. A DISPID is a well known integer value; each OLE property is identified by name or by DISPID. The value must be specified as an array of Smalltalk objects. Simple Smalltalk objects are automatically converted into their OLE equivalents, based on their classes. The following conversions are supported:

- Integer
- Float
- String
- Boolean
- OleAutomationObject

propertyAt: *aString* **withArguments**: *argumentArray*
Return the value of the named property of the OLE object. If the property is not found, this action returns nil. Otherwise, the result is the value returned by the OLE automation server. The property name must be specified as a string or a DISPID. A DISPID is a well known integer value; each OLE property is identified by name or by DISPID.

propertyAt: *aString* **withArguments:** *argumentArray* **put:** *value*
Set the value of the named property of the OLE object. If the property is not found, this action returns nil. Otherwise, the result is the value returned by the OLE automation server. The property name must be specified as a string or a DISPID. A DISPID is a well known integer value; each OLE property is identified by name or by DISPID. The value must be specified as an array of Smalltalk objects. Simple Smalltalk objects are automatically converted into their OLE equivalents, based on their classes. The following conversions are supported:

- Integer
- Float
- String
- Boolean
- OleAutomationObject

removeOleEventHandler: *eventName* **receiver:** *object* **selector:** *selector*
Remove the specified OLE event handler from the receiver. If this handler does not exist, silently do nothing.

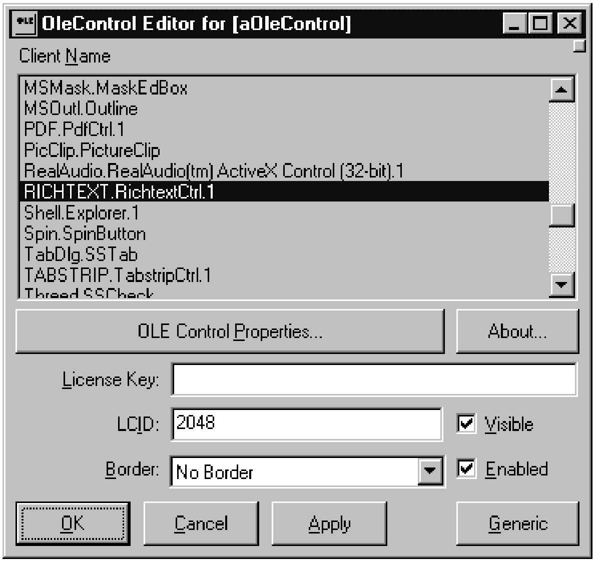
- eventName - a String or DISPID. A DISPID is a well known Integer value. Each OLE event is identified by name or by DISPID.

Callbacks & Events

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



OleControl Editor for [aOleControl]

Client Name

MSMask.MaskEdBox
MSOutl.Outline
PDF.PdfCtrl.1
PicClip.PictureClip
RealAudio.RealAudio(tm) ActiveX Control (32-bit).1
RICHTEXT.RichtextCtrl.1
Shell.Explorer.1
Spin.SpinButton
TabDlg.SSTab
TABSTRIP.TabstripCtrl.1
Threed.SSCheck

License Key:

LCID: 2048

☑ Visible

Border: No Border

☑ Enabled

About

Displays the About dialog for the selected OLE/ActiveX control.

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Client Name

The client name resource is used when the receiver is created to select the OLE class for the OLE object contained in the receiver. The OLE class name is of the form application.objecttype.version or objecttype.version where:

- application - The name of the application that supplies the object.
- objecttype - The object's name as defined in the registration database.

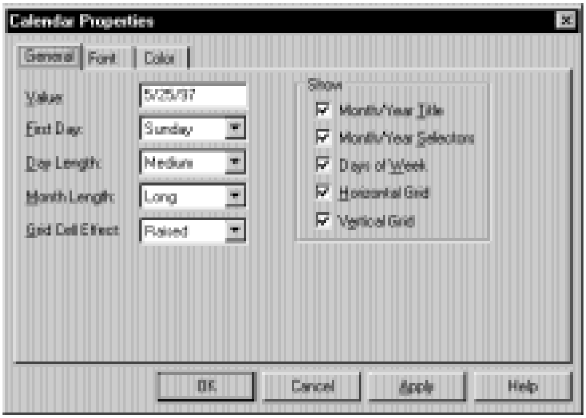
- version - The version number of the application that supplies the object.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

LCID
Advanced feature: this resource should only be set if the receiver needs its automation objects to use an LCID other than LocaleSystemDefault.

License Key
Specifies the licensing key to use when creating an OLE control. If the license key is nil, then OLE control will be created without attempting to license it. This will result in failure if licensing is required.

OLE Control Properties
This button will open up the property dialog for the selected OLE/ActiveX control. An example is shown below. Note that some properties settable via the property dialog are not code generatable. Code generatable attributes are limited to String, Integer, Float and Boolean. If you need access to all available attributes, use the file-based code generation option. Set the *Generate OLE Properties* code generation property to false.

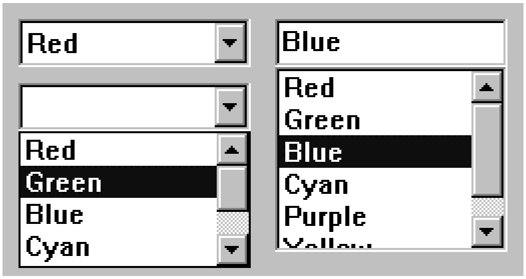


Value: 5/25/97
First Day: Sunday
Day Length: Medium
Month Length: Long
Grid Cell Effect: Raised

Show
☑ Month/Year Title
☑ Month/Year Selectors
☑ Days of Week
☑ Horizontal Grid
☑ Vertical Grid

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

WbComboBox



Combo box widgets enable the user to select from a list of available items. A combo box also displays the last selected item in a text box above the list. Combo box widgets can only have one item selected at a time.

Protocol

addItem: *item* **position:** *position*
Add an *item* to the list. This message adds an item to the list at the given *position*. A *position* value of 1 makes the first new item the first item in the list; a value of 2 makes it the second item; and so on. A value of 0 makes the first new item follow the last item in the list.

addItems: *items* **position:** *position*
Add *items* to the list. This message adds the specified items to the list at the given *position*. A *position* value of 1 makes the first new item the first item in the list; a value of 2 makes it the second item; and so on. A value of 0 makes the first new item follow the last item in the list.

comboBoxType: *anInteger*
Specifies the style of combo box.

Default: XmDROPDOWN (Drop Down)
Valid resource values:
XmSIMPLE (Simple) - The combo box always displays its list box.
XmDROPDOWN (Drop Down) - the combo box displays its list box only if the user presses the drop down button. When the button is pressed, the list box drops down, allowing the user to make a selection from the list. After the selection is made, the list disappears.

deleteAllItems
This message deletes all items from the list.

deleteItem: *item*
Delete an *item* from the list.

deleteItemsPos: *itemCount* **position:** *position*
Delete items from the list by *position*. This message deletes the specified number of items from the list starting at the specified *position*.

deletePos: *position*
Delete an item from the list by *position*. This message deletes an item at a specified *position*. A warning message appears if the position does not exist.

editable: *aBoolean*
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

getString
This message accesses the String value of the text part of the combo box.

itemCount
Specifies the total number of items in the list. It is automatically updated by the list whenever an element is added to or deleted from the list.

itemExists: *item*
Check if a specified *item* is in the list. This message is a Boolean function that checks if a specified item is present in the list.

items: *anOrderedCollection*
An array of Strings that are to be displayed as the list items.

maxLength: *anInteger*
Specifies the maximum length of the text string that can be entered into text from the keyboard.

printSelector: *aSymbol*
Specifies the selector that is used to obtain printable string representations for the items by evaluating it with each item.

replaceItemsPos: *newItems* **position:** *position*
Replace items in the list by position. This message replaces the specified number of items of the List with new items, starting at the specified position in the List. Beginning with the item specified in position, the items in the list are replaced with the corresponding elements from newItems. That is, the item at *position* is replaced with the first element of newItems; the item after *position* is replaced with the second element of newItems; and so on, until *itemCount* is reached.

selectedIndex
Answer the index of the selected item.

selectedItem
Answer the item selected in the combobox.

selectedItems
Answer an array of Strings that represents list items that are currently selected, either by the user or the application.

NOTE: For combo boxes, the collection will contain either 0 or 1 elements.

selectIndex:*itemIndex*
Select the item at *itemIndex*. Index starts at 1.

selectItem: *anObject*
Select the item *anObject*. *anObject* can be an index or a string.

setString: *value*
This message sets the string *value* of the text part of the combo box.

verifyBell: *aBoolean*
Specifies whether the bell should sound when the verification returns without continuing the action.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Activate Callback
These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback
These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

Call data arguments:

text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

Popdown Callback

These callbacks are triggered when the item list disappears

Popup Callback

These callbacks are triggered when the item list appears

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Single Selection Callback

These callbacks are triggered when the user selects an item in the list, or presses an arrow key to move through the list.

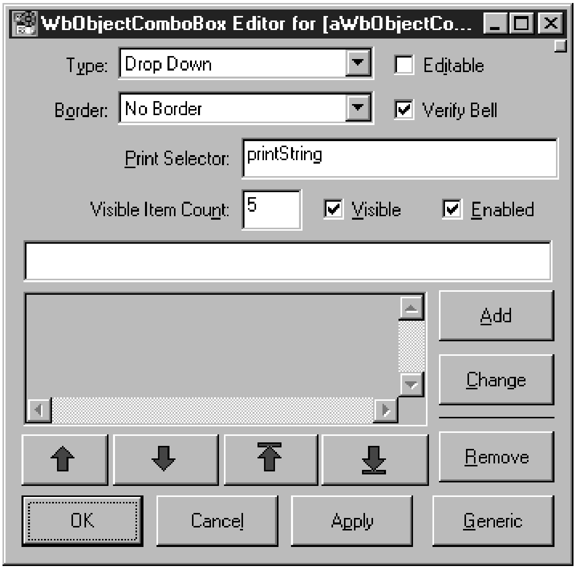
Call data arguments:

item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.

Value Changed Callback

These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



WbObjectComboBox Editor for [aWbObjectCo...

Type: Drop Down

□ Editable

Border: No Border

☑ Verify Bell

Print Selector: printString

Visible Item Count: 5

☑ Visible

☑ Enabled

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

> Border - Causes the widget to have a border.
> No Border - Causes the widget to have no border.

Combo Box Type
Specifies the style of combo box.

> Drop Down - the combo box displays its list box only if the user presses the drop down button. When the button is pressed, the list box drops down, allowing the user to make a selection from the list. After the selection is made, the list disappears.
> Simple - The combo box always displays its list box.

Editable
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Items
An array of Strings that are to be displayed as the list items.

Print Selector
Specifies the selector that is used to obtain printable string representations for the items by evaluating it.

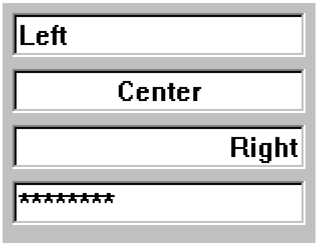
Max Length
Specifies the maximum length of the text string that can be entered into text from the keyboard.

Verify Bell
Specifies whether the bell should sound when the verification returns without continuing the action.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Visible Item Count
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

WbEnhancedText



WbEnhancedText widgets are advanced single line entry fields that provide character and field level validation. Character level validations include: Alpha, AlphaNumeric, Boolean, Integer, and PositiveInteger. Field level validations include: Date, PhoneNumberUS, SSN, ZipCodeUS, etc.

In addition to validation functions, special behaviors may be specified for gaining and losing focus. Additional styles such as *password* are available. Alignment (right, left, centered) may be specified.

Additional validation functions may be added easily. All of the validation functions are public methods of the WbEnhancedText class and begin with the characters ‘ok’. Character level validations take one argument - the character itself. The routine should respond with true or false depending on whether the character is valid. For example, to create a validation function that would only allow the asterisk character to be entered, the following code would be required:

```
okAsteriskOnly: aChar
   ^aChar == $*
```

Field level validations take no arguments and work on the entire contents of the field. They should Answer true or false depending on whether the field in valid. They may also optionally reformat the field as required. A simple field level validation that would test whether the contents is a palindrome could be coded as follows:

```
okPalindrome
   ^self contents = self contents reversed
```

The attribute editor automatically displays all validation functions that it identifies within the WbEnhancedText class.

Protocol

alignment: *anInteger*
Specifies the text alignment used by the widget.

Default: XmALIGNMENTBEGINNING (Left)
Valid resource values:
XmALIGNMENTBEGINNING (Left) - Causes the left side of the line of text to be vertically aligned with the left edge of the widget window.
XmALIGNMENTCENTER (Center) - Causes the center of the line of text to be vertically aligned in the center of the widget window.
XmALIGNMENTEND (Right) - Causes the right side of the line of text to be vertically aligned with the right edge of the widget window.

case: *aSymbol*
Specifies the case of the text in the widget.

Default: #Unchanged (Unchanged)
Valid resource values:
#lower (lower) - Lower case.
#Proper (Proper) - Proper case (first letters of each word uppercase).
#Unchanged (Unchanged) - Unchanged case.
#UPPER (UPPER) - Upper case.

character: *aSymbol*
Specifies the function used to validate each character as it is entered.

Default: #okAny: (Any)
Valid resource values:
#ok20Comma10: (20Comma10) - Validates whether each character is a valid for a 20,10 field.
#ok7Comma2: (7Comma2) - Validates whether each character is a valid for a 7,2 field.
#ok7Comma4: (7Comma4) - Validates whether each character is a valid for a 7,4 field.
#okAlpha: (Alpha) - Validates whether each character is a valid alpha character ($A - $Z, $a - $z or space).
#okAlphaNoSpace: (AlphaNoSpace) - Validates whether each character is a valid alpha character ($A - $Z, $a - $z).
#okAlphaNumeric: (AlphaNumeric) - Validates whether each character is a valid alpha-numeric character ($A - $Z, $a - $z, $0 - $9, or space).
#okAlphaNumericNoSpace: (AlphaNumericNoSpace) - Validates whether each character is a valid alpha character ($A - $Z, $a - $z, $0 - $9).
#okAny: (Any) - Any character is valid.
#okBoolean: (Boolean) - Validates whether each character is a valid Boolean value (e.g., T, t, F, f, Y, y, N, or n).
#okInteger: (Integer) - Validates whether each character is a valid Integer value.
#okNumeric: (Numeric) - Validates whether each character is a valid numeric value.

#okPositive10Comma10: (Positive10Comma10) - Validates whether each character is a valid for a positive 10,10 field.
#okPositiveInteger: (PositiveInteger) - Validates whether each character is a valid positive Integer.
#okPositiveNumeric: (PositiveNumeric) - Validates whether each character is a valid positive numeric value.

clearSelection
Clear the selection.

columns: *anInteger*
Specifies the initial width of the text window measured in character spaces.

copySelection
Copy the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

cursorPosition: *anInteger*
Indicates the position in the text where the current insert cursor is to be located. Position is determined by the number of characters from the beginning of the text.

cutSelection
Cut the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

editable: *aBoolean*
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

editMode: *anInteger*
Specifies whether the widget supports single line or multi line editing of text.

Default: XmSINGLELINEEDIT (Single Line)
Valid resource values:
XmMULTILINEEDIT (Multi Line) - Multi line text edit.
XmSINGLELINEEDIT (Single Line) - Single line text edit.

field: *aSymbol*
Specifies the function used to validate/format the contents of the widget when it looses focus.

Default: #okAny (Any)
Valid resource values:
#okAny (Any) - Any characters are valid.
#okCurrency (Currency) - Formats the contents of the field to be a currency value.
#okCurrencyNoDecimal (CurrencyNoDecimal) - Formats the contents of the field to be a currency value without decimal places.

#okDate (Date) - Validates whether the contents of the field are a valid Date value and then formats the Date using the system Date format.
#okInteger (Integer) - Formats the contents of the field to be an Integer value.
#okNumber (Number) - Formats the contents of the field to be a numeric (e.g., Float) value.
#okPhoneNumberExtUS (PhoneNumberExtUS) - Validates and formats the contents of the field to be a US phone number with optional extension. Valid formats are: (999) 999-9999, 999-9999, x999, (999) 999-9999 x999.
#okPhoneNumberUS (PhoneNumberUS) - Validates and formats the contents of the field to be a US phone number. Valid formats are: (999) 999-9999, 999-9999.
#okRound2 (Round2) - Formats the contents of the field to be rounded to 2 decimal places.
#okRound3 (Round3) - Formats the contents of the field to be rounded to 3 decimal places.
#okSSN (SSN) - Validates and formats the contents of the field to be a US Social Security Number. Format is: 999-99-9999.
#okZipCodeUS (ZipCodeUS) - Validates and formats the contents of the field to be a US Zip Code. Valid format are: 99999 or 99999-9999.

getEditable
This message accesses the edit permission state of the Text widget.

getFocus: *aSymbol*
Specifies the cursor location when the widget gains focus.

Default: #selectAll (Select All)
Valid resource values:
#selectAll (Select All) - Select all text on gaining focus.
#selectFirst (Select First) - Place cursor at beginning of text when gaining focus.
#selectLast (Select Last) - Place cursor at end of text when gaining focus.

getInsertionPosition
Return the position of the insert cursor. The return value is an integer number of characters from the beginning of the text buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getLastPosition
This message returns an Integer value that indicates the position of the last character in the text buffer. This is an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getSelection
Return a String containing the selection, or nil if there is no selection.

getSelectionPosition
Return a Point describing the selection position, where the x value is the start of the selection, and the y value is the end of the selection. The positions are an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getString
This message accesses the String value of the Text widget.

getTopCharacter
This message returns an Integer value that indicates the number of characters from the beginning of the text buffer. The first character position is 0.

insert: *position* **value:** *value*
Insert a String into the text. This message inserts a character string into the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This routine also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

insertAndShow: *position* **value:** *value*
Insert a String into the text and ensure that the text widget is scrolled such that the line containing the last new character inserted is visible. Vertical and/or horizontal scrolling may occur to accomplish this. This specification does not require that the text widget scroll horizontally but allows it. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This message also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

largeText: *aBoolean*
This is a hint that indicates that the receiver will be processing a large amount of text. If this flag is false, text operations may fail due to space limitations. If this hint is true, text operations will not fail.

lineDelimiter
Answer a String containing the line delimiting sequence used by the receiver. This value and number of characters may vary from platform to platform. The sequence is usually the standard end of line sequence for the platform. For example, on X/MOTIF this value is a String containing an ASCII LF character. On Windows, this value is a String containing ASCII CR and LF characters. All computations involving text positions operate consistently with the number of characters in the lineDelimiter String. Thus an end of line takes up 1 character position on X and 2 character positions on Windows.

maxLength: *anInteger*
Specifies the maximum length of the text string that can be entered into text from the keyboard.

paste
Insert the clipboard selection into the text. Answer true if the operation is successful, or false if the text could not be retrieved from the clipboard.

password: *aBoolean*
Specifies whether the widget use the password style.

remove
Delete the selection.

replace: *fromPos* **toPos:** *toPos* **value:** *value*
Replace part of the receiver’s text String. This message replaces part of the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. An example text replacement would be to replace the second and third characters in the text string. To accomplish this, the parameter fromPos must be 1 and toPos must be 3. To insert a string after the fourth character, both parameters, *fromPos* and *toPos*, must be 4. This message also calls the widget’s XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

rows: *anInteger*
Specifies the initial height of the text window measured in character heights.

scroll: *lines*
Scroll the text. This message scrolls text in a Text widget. *lines* specifies the number of lines of text to scroll. A positive value causes text to scroll upward; a negative value causes text to scroll downward.

scrollHorizontal: *aBoolean*
Adds a ScrollBar that allows the user to scroll horizontally through text.

scrollVertical: *aBoolean*
Adds a ScrollBar that allows the user to scroll vertically through text.

setEditable: *aBoolean*
This message sets the edit permission state of the Text widget. When set to True, the text string can be edited.

setHighlight: *positions* **mode:** *mode*
Set the text highlight. This message sets highlights text between the two specified character positions. The mode parameter determines the type of highlighting. Highlighting text merely changes the visual appearance of the text; it does not set the selection.

setInsertionPosition: *position*
Set the *position* of the insert cursor. This message sets the insertion cursor position of the Text widget.

setSelection: *positions*
Set the selection. This message sets the primary selection of the text in the widget. It also sets the insertion cursor position to the last position of the selection.

setString: *value*
This message sets the string value of the Text widget.

setTopCharacter: *topCharacter*
This message sets the position of the text at the top of the Text widget. If the editMode is XmMULTILINEEDIT, the line of text that contains *topCharacter* is displayed at the top of the widget without shifting the text left or right.

showPosition: *position*
Force text at the specified position to be displayed. This message forces text at the specified position to be displayed.

tabSpacing: *anInteger*
Indicates the tab stop spacing.

topCharacter: *anInteger*
Displays the position of text at the top of the window. Position is determined by the number of characters from the beginning of the text.

twoDigitYear: *anInteger*
Specifies the interpretation of a two digit year.

value: *aString*
Specifies the displayed text String.

verifyBell: *aBoolean*
Specifies whether the bell should sound when the verification returns without continuing the action.

wordWrap: *aBoolean*
Indicates that lines are to be broken at word breaks (i.e., the text does not go off the right edge of the window).

Callbacks & Events

Activate Callback
These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Focus Callback
These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback
These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback
These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

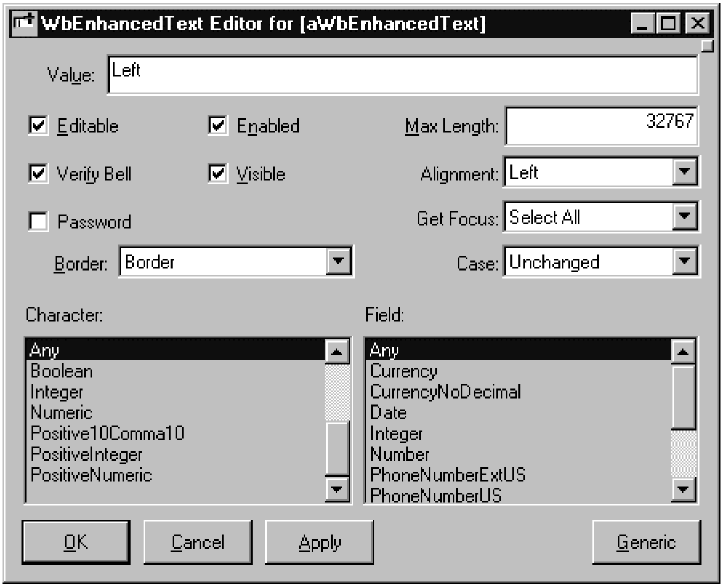
Call data arguments:
text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

Resize Callback
These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback

These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



WbEnhancedText Editor for [aWbEnhancedText]

Value: Left

☑ Editable ☑ Enabled Max Length: 32767

☑ Verify Bell ☑ Visible Alignment: Left

☐ Password Get Focus: Select All

Border: Border Case: Unchanged

Alignment

Specifies the text alignment used by the widget.

Left - Causes the left side of the line of text to be vertically aligned with the left edge of the widget window.
Center - Causes the center of the line of text to be vertically aligned in the center of the widget window.
Right - Causes the right side of the line of text to be vertically aligned with the right edge of the widget window.

Border Width

Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Case
Specifies the case of the text in the widget.

lower - Lower case.
Proper - Proper case (first letters of each word uppercase).
Unchanged - Unchanged case.
UPPER - Upper case.

Character
Specifies the function used to validate each character as it is entered.

Alpha
Only the characters $A-$Z, $a-$z, and space are allowed.

AlphaNoSpace
Only the characters $A-$Z and $a-$z are allowed.

AlphaNumeric
Only alpha characters, the digits $0-$9, and space are allowed.

AlphaNumericNoSpace
Only alpha characters and the digits $0-$9 are allowed.

Any
Any character is allowed

Boolean
Only the characters $T, $t, $F, $f, $Y, $y, $N, and $n are allowed.

Integer
Only acceptable integers (positive or negative) are allowed.

Numeric
Only acceptable numbers (positive or negative) are allowed.

PositiveInteger
Only the digits $0-$9 are allowed.

PositiveNumeric
Only positive numbers are allowed .

Positive10Comma10
Only positive numbers with up to 10 digits before the decimal and 10 after the decimal are allowed .

20Comma10
Only acceptable numbers with up to 20 digits before the decimal and 10 after the decimal are allowed .

7Comma2
Only acceptable numbers with up to 7 digits before the decimal and 2 after the decimal are allowed .

7Comma4
Only acceptable numbers with up to 7 digits before the decimal and 4 after the decimal are allowed .

Editable

Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Field

Specifies the function used to validate/format the contents of the widget when it looses focus.

Any
Any characters are valid.

Currency
The contents of the entry field are reformatted as a standard currency value using the National Language currency format and the default number of digits.

CurrencyNoDecimal
The contents of the entry field are reformatted as a standard currency value using the National Language currency format and no decimal places.

Date
The contents of the field must be a valid date. A variety of formats are supported. The field is automatically reformatted to reflect the system date format.

Integer
Formats the contents of the field to be an Integer value.

Number
Formats the contents of the field to be a numeric (e.g., Float) value.

PhoneNumberExtUS
The contents of the entry field are reformatted as a standard US phone number complete with extension. Any Alpha characters will be converted to there phone number equivalents.

PhoneNumberUS
The contents of the entry field are reformatted as a standard US phone number. An error is generated if the field contains other than seven or ten characters. Any Alpha characters will be converted to there phone number equivalents.

Round2
Round the contents of the field to two decimal places.

Round3
Round the contents of the field to three decimal places.

SSN
The contents must be a valid Social Security Number. The contents will be automatically reformatted.

ZipCodeUS
The contents must be a valid five or nine character US zip code. The contents will be reformatted if required.

Get Focus
Specifies the cursor location when the widget gains focus.

Select All - Select all text on gaining focus.
Select First - Place cursor at beginning of text when gaining focus.
Select Last - Place cursor at end of text when gaining focus.

Max Length
Specifies the maximum length of the text string that can be entered into text from the keyboard.

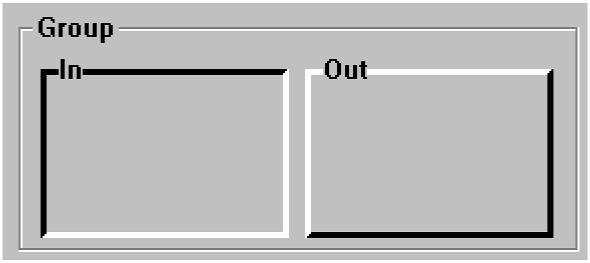
Password
Specifies whether the widget use the password style.

Value
Specifies the displayed text String.

Verify Bell
Specifies whether the bell should sound when the verification returns without continuing the action.

Visible
Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

WbFrame



Frame widgets are used to visually indicate and label groups of related controls. They are composed of a box with an optional label in the upper left corner. WbFrames allow you to specify the frame thickness for the in and out styles.

***Note:** Frame widgets in VisualAge Smalltalk are allowed to have one and only one child. In order for a frame to group multiple widgets, a form widget should be inserted into the frame as its sole child. The form then acts as the parent of any other widgets placed within the bounds of the frame. The **Always Add Forms To Frames** command will cause a form to be automatically inserted into any new frame.*

Protocol

frameThickness: *anInteger*
Specifies the frame thickness when the widget is in *in* or *out* mode.

labelString: *aString*
Specifies the label string.

marginHeight: *anInteger*
Specifies the padding space on the top and bottom sides between the child of Frame and Frame’s shadow drawing.

marginWidth: *anInteger*
Specifies the padding space on the left and right sides between the child of Frame and Frame’s shadow drawing.

shadowType: *anInteger*
Describes the drawing style for Frame.

Default: XmSHADOWDEFAULT (Default)
Valid resource values:

XmSHADOWDEFAULT (Default) - Draws Frame in a platform specific manner.
XmSHADOWETCHEDIN (Etched In) - Draws Frame using a double line giving the effect of a line etched into the window.

XmSHADOWETCHEDOUT (Etched Out) - Draws Frame using a double line giving the effect of a line coming out of the window.
XmSHADOWIN (In) - Draws Frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
XmSHADOWOUT (Out) - Draws Frame such that it appears outset.

Callbacks & Events

Expose Callback

These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback

These callbacks are triggered when any area of the widget or its children is exposed.

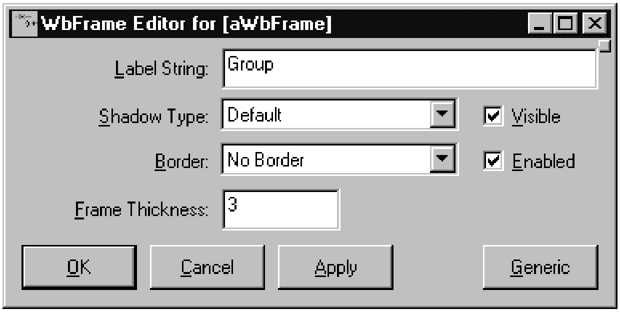
Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Editor



WbFrame Editor for [aWbFrame]

Label String: Group
Shadow Type: Default
Border: No Border
Frame Thickness: 3
☑ Visible
☑ Enabled

Border Width

Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Frame Thickness

Specifies the frame thickness when the widget is in *in* or *out* mode.

Label String

Specifies the label string.

Shadow Type

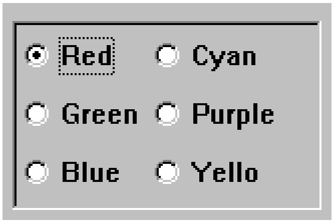
Describes the drawing style for Frame.

Default - Draws Frame in a platform specific manner.
Etched In - Draws Frame using a double line giving the effect of a line etched into the window.
Etched Out - Draws Frame using a double line giving the effect of a line coming out of the window.
In - Draws Frame such that it appears inset. This means that the bottom shadow visuals and top shadow visuals are reversed.
Out - Draws Frame such that it appears outset.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

WbRadioBox



RadioBoxes provide a mechanism for the user to select one mutually exclusive option from a set of options.

Protocol

adjustLast: *aBoolean*
Extends the last row of children to the bottom edge of RowColumn (when orientation is horizontal) or extends the last column to the right edge of RowColumn (when orientation is vertical). This feature is disabled by setting XmNadjustLast to false.

buttonSet: *anInteger*
Specifies which button of a RadioBox or OptionMenu Pulldown submenu is initially set. The value is an integer n indicating the nth ToggleButton specified for a RadioBox or the nth PushButton specified for an OptionMenu Pulldown submenu. The first button specified is number 0.

isAligned: *aBoolean*
Specifies text alignment for each item within the RowColumn widget; this only applies to items which are a subclass of CwLabel, and on some platforms, applies only to instances of CwLabel.

isHomogeneous: *aBoolean*
Indicates if the RowColumn widget should enforce exact homogeneity among the items it contains.

items: *anOrderedCollection*
Specifies a collection of strings to use as labels for the buttons created.

marginHeight: *anInteger*
Specifies the amount of blank space between the top edge of the RowColumn widget and the first item in each column, and the bottom edge of the RowColumn widget and the last item in each column.

marginWidth: *anInteger*
Specifies the amount of blank space between the left edge of the RowColumn widget and the first item in each row, and the right edge of the RowColumn widget and the last item in each row.

numColumns: *anInteger*
For vertically-oriented RowColumn widgets, this attribute indicates how many columns are built; the number of entries per column are adjusted to maintain this number of columns, if possible. For horizontally-oriented RowColumn widgets, this attribute indicates how many rows are built.

orientation: *anInteger*
Determines whether RowColumn layouts are row major or column major.

Default: XmVERTICAL (Column)
Valid resource values:

XmVERTICAL (Column) - In a column major layout, the children of the RowColumn are laid out in columns top to bottom within the widget.

XmHORIZONTAL (Row) - In a row major layout the children of the RowColumn are laid out in rows left to right within the widget.

packing: *anInteger*
Specifies how to pack the items contained within a RowColumn widget.

Default: XmPACKTIGHT (Pack Tight)
Valid resource values:

XmPACKTIGHT (Pack Tight) - Indicates that given the current major dimension, entries are placed one after the other until the RowColumn widget must wrap.

XmPACKCOLUMN (Pack Column) - Indicates that all entries are placed in identically sized boxes.

XmPACKNONE (Pack None) - Indicates that no packing is performed.

radioAlwaysOne: *aBoolean*
Forces the active ToggleButton to be automatically selected after having been unselected (if no other toggle was activated), if true. If false, the active toggle may be unselected. The default value is true.

radioBehavior: *aBoolean*
Specifies that the RowColumn widget should enforce a RadioBox-type behavior on all of its children which are ToggleButtons.

resizeHeight: *aBoolean*
Requests a new height if necessary, when set to true. When set to false, the widget does not request a new height regardless of any changes to the widget or its children.

resizeWidth: *aBoolean*
Requests a new width if necessary, when set to true. When set to false, the widget does not request a new width regardless of any changes to the widget or its children.

spacing: *anInteger*
Specifies the horizontal and vertical spacing between items contained within the RowColumn widget.

Callbacks & Events

Entry Callback

Supply a single callback routine for handling all items contained in a RowColumn widget. This disables the activation callbacks for all ToggleButton and PushButton widgets contained within the RowColumn widget.

Call data arguments:
widget - the value of widget. This is the widget that triggered the Entry Callback.
callbackData - the callData from the widget that triggered the Entry Callback.
data - the value of data.

Expose Callback

These callbacks are triggered when the widget receives an exposure event requiring it to repaint itself.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Intercept Expose Callback

These callbacks are triggered when any area of the widget or its children is exposed.

Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Map Callback

These callbacks are triggered when the window associated with the RowColumn widget is about to be mapped.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

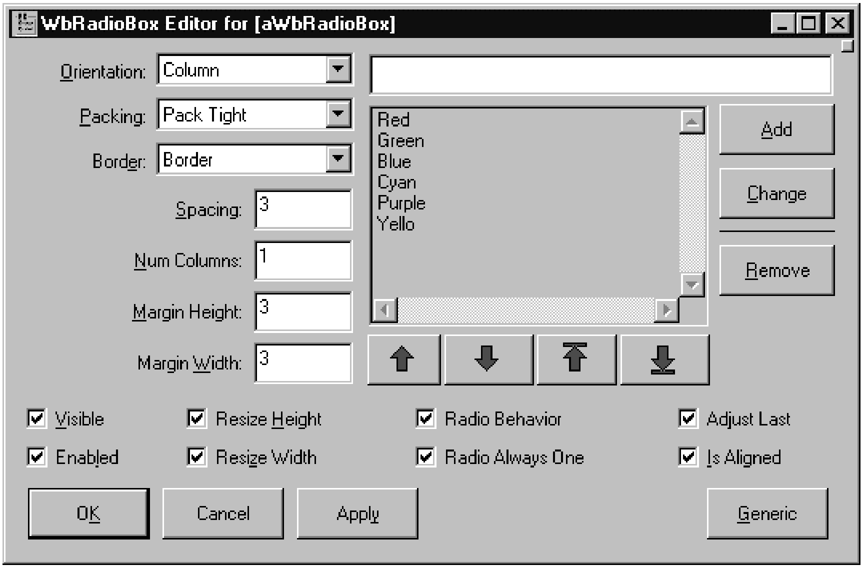
Simple Callback

These callbacks are triggered when a button is activated or when its value changes.

Unmap Callback

These callbacks are triggered after the window associated with the RowColumn widget has been unmapped.

Editor



WbRadioBox Editor for [aWbRadioBox]

Orientation: Column

Packing: Pack Tight

Border: Border

Spacing: 3

Num Columns: 1

Margin Height: 3

Margin Width: 3

Red
Green
Blue
Cyan
Purple
Yello

☑ Visible ☑ Resize Height ☑ Radio Behavior ☑ Adjust Last

☑ Enabled ☑ Resize Width ☑ Radio Always One ☑ Is Aligned

Adjust Last

Extends the last row of children to the bottom edge of RadioBox (when orientation is horizontal) or extends the last column to the right edge of RadioBox (when orientation is vertical).

Border Width

Specifies the width of the border that surrounds the widget’s window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

> Border - Causes the widget to have a border.
> No Border - Causes the widget to have no border.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Is Aligned

Specifies text alignment for each item within the RadioBox widget; this only applies to items which are a subclass of CwLabel, and on some platforms, applies only to instances of CwLabel.

Items
Specifies a collection of strings to use as labels for the buttons created.

Margin Height
Specifies the amount of blank space between the top edge of the RadioBox widget and the first item in each column, and the bottom edge of the RadioBox widget and the last item in each column.

Margin Width
Specifies the amount of blank space between the left edge of the RadioBox widget and the first item in each row, and the right edge of the RadioBox widget and the last item in each row.

Num Columns
For vertically-oriented RadioBox widgets, this attribute indicates how many columns are built; the number of entries per column are adjusted to maintain this number of columns, if possible. For horizontally-oriented RadioBox widgets, this attribute indicates how many rows are built.

Orientation
Determines whether RadioBox layouts are row major or column major.

Column - In a column major layout, the children of the RadioBox are laid out in columns top to bottom within the widget.
Row - In a row major layout the children of the RadioBox are laid out in rows left to right within the widget.

Packing
Specifies how to pack the items contained within a RadioBox widget.

Pack Column - Indicates that all entries are placed in identically sized boxes.
Pack None - Indicates that no packing is performed.
Pack Tight - Indicates that given the current major dimension, entries are placed one after the other until the RadioBox widget must wrap.

Radio Always One
Forces the active ToggleButton to be automatically selected after having been unselected (if no other toggle was activated), if true. If false, the active toggle may be unselected. The default value is true.

Radio Behavior
Specifies that the RadioBox widget should enforce a RadioBox-type behavior on all of its children which are ToggleButtons.

Resize Height
Requests a new height if necessary, when set to true. When set to false, the widget does not request a new height regardless of any changes to the widget or its children.

Resize Width

Requests a new width if necessary, when set to true. When set to false, the widget does not request a new width regardless of any changes to the widget or its children.

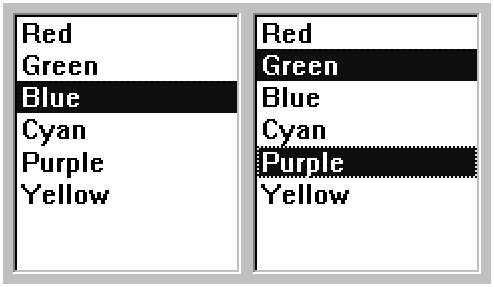
Spacing

Specifies the horizontal and vertical spacing between items contained within the RowColumn widget.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

WbScrolledList



List widgets present a list of items and allow the user to select one or more items from the list. List widgets provide several methods for adding, deleting and replacing items and selected items in the list. The selectionPolicy resource specifies the policy for selecting items. Four selection modes are supported: browse select, single select, multiple select and extended select.

Protocol

addItem: *item* **position:** *position*
Add an *item* to the list. This message adds an item to the list at the given *position*. When the item is inserted into the list, it is compared with the current selectedItems list. If the new item matches an item on the selected list, it appears selected.

addItems: *items* **position:** *position*
Add *items* to the list. This message adds the specified items to the list at the given *position*. When the items are inserted into the list, they are compared with the current selectedItems list. If the any of the new items matches an item on the selected list, it appears selected.

addItemUnselected: *item* **position:** *position*
Add an *item* to the list, forcing it to be unselected. This message adds an *item* to the list at the given *position*. The item does not appear selected, even if it matches an item in the current selectedItems list.

deleteAllItems
This message deletes all items from the list.

deleteItem: *item*
Delete an *item* from the list.

deleteItems: *items*
This message deletes the specified *items* from the list. A warning message appears if any of the items do not exist.

deleteItemsPos: *itemCount* **position:** *position*
Delete items from the list by *position*. This message deletes the specified number of items from the list starting at the specified position.

deletePos: *position*
Delete an item from the list by *position*. This message deletes an item at a specified position. A warning message appears if the position does not exist.

deselectAllItems
Unhighlight and remove all elements from the selectedItems list.

deselectItem: *item*
Unhighlight and remove the specified *item* from the selected list.

deselectPos: *position*
Unhighlight and remove an item from the selected list by *position*.

getMatchPos: *item*
This message returns an Array of Integer positions where a specified *item* is found in a List. If the item does not occur in the list the resulting Array is empty. The #= operator is used for the search.

getSelectedPos
Return an Array containing the positions of every selected item in the list.

itemCount
Answer the total number of items. It is automatically updated by the list whenever an element is added to or deleted from the list.

itemExists: *item*
Check if a specified *item* is in the list. This message is a Boolean function that checks if a specified item is present in the list. The #= operator is used for the search.

items: *anOrderedCollection*
An array of Strings that are to be displayed as the list items.

replaceItems: *oldItems* **newItems:** *newItems*
This message replaces each specified item of the list with a corresponding new item. Every occurrence of each element of *oldItems* is replaced with the corresponding element from *newItems*. That is, the first element of *oldItems* is replaced with the first element of

newItems. The second element of *oldItems* is replaced with the second element of *newItems*, and so on. The #= operator is used for the search.

replaceItemsPos: *newItems* **position:** *position*
Replace items in the list by position. This message replaces the specified number of items of the List with new items, starting at the specified *position* in the List. Beginning with the item specified in *position*, the items in the list are replaced with the corresponding elements from *newItems*. That is, the item at *position* is replaced with the first element of *newItems*; the item after *position* is replaced with the second element of *newItems*; and so on, until itemCount is reached.

scrollHorizontal: *aBoolean*
This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

selectedItemCount
Answer the number of strings in the selected items list.

selectedIndex
Answer the index of the selected item.

selectedItem
Answer the item selected in the list.

selectedItems: *anOrderedCollection*
An array of Strings that represents the list items that are currently selected, either by the user or the application.

selectIndex:*itemIndex*
Select the item at *itemIndex*. Index starts at 1.

selectItem: *anObject*
Select the item *anObject*. *anObject* can be an index or a string.

selectionPolicy: *anInteger*
Defines the interpretation of the selection action.

Default: XmBROWSESELECT (Browse Select)
Valid resource values:

XmSINGLESELECT (Single Select) - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

XmMULTIPLESELECT (Multiple Select) - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

XmEXTENDEDSELECT (Extended Select) - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

XmBROWSESELECT (Browse Select) - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

selectItem: *item* **notify:** *notify*
Select an *item* in the list. This message highlights and adds the specified *item* to the current selected list. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

selectPos: *position* **notify:** *notify*
Select an item in the list by *position*. This message highlights a List item at the specified *position* and adds it to the list of selected items. *notify* specifies a Boolean value that when true invokes the selection callback for the current mode. From an application interface view, calling this function with notify true is indistinguishable from a user initiated selection action.

setBottomItem: *item*
Make an existing *item* the last visible item in the list. This message makes an existing item the last visible item in the list. The item can be any valid item in the list.

setBottomPos: *position*
Make an item the last visible item in the list by *position*. This message makes the item at the specified *position* the last visible item in the List.

setItem: *item*
Make an existing *item* the first visible item in the list. This message makes an existing item the first visible item in the list. The item can be any valid item in the list.

setPos: *position*
Make an item the first visible item in the list by *position*. This message makes the item at the given *position* the first visible position in the List.

topItemPosition: *anInteger*
Specifies the position of the item that is the first visible item in the list.

visibleItemCount: *anInteger*
Specifies the number of items that can fit in the visible space of the List work area. The list will use this value to determine its height.

Callbacks & Events

Browse Selection Callback

These callbacks are triggered when an item is selected in the browse selection mode. It is only valid when Selection Policy is Browse Select.

Call data arguments:
item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.

Default Action Callback

These callbacks are triggered when an item is double clicked.

Call data arguments:
item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.

Extended Selection Callback

These callbacks are triggered when items are selected using the extended selection mode. It is only valid when Selection Policy is Extended Select.

Call data arguments:
item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of Strings which are the selected items.

Multiple Selection Callback

These callbacks are triggered when an item is selected in multiple selection mode. It is only valid when Selection Policy is Multiple Select.

Call data arguments:
item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.
selectedItemCount - the integer number of selected items.
selectedItemPositions - a Collection containing the integer positions of the selected items.
selectedItems - a Collection of Strings which are the selected items.

Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

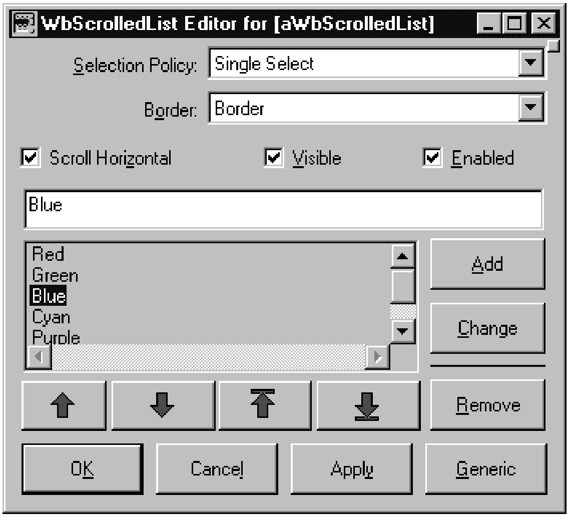
Single Selection Callback

These callbacks are triggered when an item is selected in single selection mode. It is only valid when Selection Policy is Single Select.

Call data arguments:

item - the String which is the selected item.
itemPosition - the integer position of the selected item in the list.

Editor



WbScrolledList Editor for [aWbScrolledList]

Selection Policy: Single Select

Border: Border

☑ Scroll Horizontal ☑ Visible ☑ Enabled

Blue

Red
Green
Blue
Cyan
Purple

Border Width
Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Enabled
Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Items
An array of Strings that are to be displayed as the list items.

Scroll Horizontal
This resource is a hint that a horizontal scroll bar is desired for this list. The hint is ignored on platforms where the feature is not configurable.

Selected Items
An array of Strings that represents the list items that are currently selected, either by the user or the application.

Selection Policy

Defines the interpretation of the selection action.

Browse Select - Allows only single selection. The selection changes when the mouse is dragged. This is the default Selection Policy. Under Windows and OS/2, this is the same as Single Select.

Extended Select - Allows multiple items to be selected, either by dragging the selection or by clicking on items with a modifier key held down. Clicking on an item without a modifier key held down deselects all previously selected items.

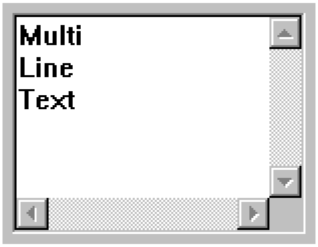
Multiple Select - Allows multiple items to be selected. The selection of an item is toggled when it is clicked on. Clicking on an item does not deselect previously selected items.

Single Select - Allows only single selections. Under Windows and OS/2, this is the same as Browse Select.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

WbScrolledText


Multi
Line
Text

Scrolled text widgets provide multi-line text viewing and editing capabilities to the application. If the user types more text than can be accommodated within the field, it will automatically scroll.

Protocol

clear
Clear the contents of the receiver.

clearSelection
Clear the selection.

columns: *anInteger*
Specifies the initial width of the text window measured in character spaces.

copySelection
Copy the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

cursorPosition: *anInteger*
Indicates the position in the text where the current insert cursor is to be located. Position is determined by the number of characters from the beginning of the text.

cutSelection
Cut the selection to the clipboard. Answer true if the operation is successful, or false if the text could not be placed in the clipboard.

editable: *aBoolean*
Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

editMode: *anInteger*
Specifies whether the widget supports single-line or multi-line editing of text.

Default: XmMULTILINEEDIT (Multi Line)
Valid resource values:
    XmMULTILINEEDIT (Multi Line) - Multi-line text edit.
    XmSINGLELINEEDIT (Single Line) - Single-line text edit.

getEditable
This message accesses the edit permission state of the Text widget.

getInsertionPosition
Return the position of the insert cursor. The return value is an integer number of characters from the beginning of the text buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getLastPosition
This message returns an Integer value that indicates the position of the last character in the text buffer. This is an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getSelection
Return a String containing the selection, or nil if there is no selection.

getSelectionPosition
Return a Point describing the selection position, where the x value is the start of the selection, and the y value is the end of the selection. The positions are an integer number of characters from the beginning of the buffer. The first character position is 0.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

getString
This message accesses the String value of the Text widget.

getTopCharacter
This message returns an Integer value that indicates the number of characters from the beginning of the text buffer. The first character position is 0.

insert: *position* **value:** *value*
Insert a String into the text. This message inserts a character string into the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This routine also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

insertAndShow: *position* **value:** *value*
Insert a String into the text and ensure that the text widget is scrolled such that the line containing the last new character inserted is visible. Vertical and/or horizontal scrolling may occur to accomplish this. This specification does not require that the text widget scroll horizontally but allows it. The character positions begin at zero and are numbered sequentially from the beginning of the text. For example, to insert a string after the fourth character, the parameter *position* must be 4. This message also calls the widget's XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

NOTE: Some platforms use a 1-character line delimiter and some use a 2-character line delimiter. The position value is consistent with the standard platform line delimiter sequence.

largeText: *aBoolean*
This is a hint that indicates that the receiver will be processing a large amount of text. If this flag is false, text operations may fail due to space limitations. If this hint is true, text operations will not fail.

lineDelimiter
Answer a String containing the line delimiting sequence used by the receiver. This value and number of characters may vary from platform to platform. The sequence is usually the standard end of line sequence for the platform. For example, on X/MOTIF this value is a String containing an ascii LF character. On Windows, this value is a String containing ascii CR and LF characters. All computations involving text positions operate consistently with the number of characters in the lineDelimiter String. Thus an end of line takes up 1 character position on X and 2 character positions on Windows.

maxLength: *anInteger*
Specifies the maximum length of the text string that can be entered into text from the keyboard.

paste
Insert the clipboard selection into the text. Answer true if the operation is successful, or false if the text could not be retrieved from the clipboard.

readOnly
Set the readonly property of the receiver.

readWrite
Clear the readonly property of the receiver.

remove
Delete the selection.

replace: *fromPos* **toPos:** *toPos* **value:** *value*
Replace part of the receiver’s text String. This message replaces part of the text string in the Text widget. The character positions begin at zero and are numbered sequentially from the beginning of the text. An example text replacement would be to replace the second and third characters in the text string. To accomplish this, the parameter fromPos must be 1 and toPos must be 3. To insert a string after the fourth character, both parameters, *fromPos* and *toPos*, must be 4. This message also calls the widget’s XmNmodifyVerifyCallback and XmNvalueChangedCallback callbacks.

rows: *anInteger*
Specifies the initial height of the text window measured in character heights.

scroll: *lines*
Scroll the text. This message scrolls text in a Text widget. *lines* specifies the number of lines of text to scroll. A positive value causes text to scroll upward; a negative value causes text to scroll downward.

scrollHorizontal: *aBoolean*
Adds a ScrollBar that allows the user to scroll horizontally through text.

scrollVertical: *aBoolean*
Adds a ScrollBar that allows the user to scroll vertically through text.

selectAll
Select the entire text of the receiver.

selectAtEnd
Place the gap selection at the end of the text.

selectedItem
Answer a String containing the text selected in clipboard format.

setEditable: *aBoolean*
This message sets the edit permission state of the Text widget. When set to True, the text string can be edited.

setHighlight: *positions* **mode:** *mode*
Set the text highlight. This message highlights text between the two specified character positions. The mode parameter determines the type of highlighting. Highlighting text merely changes the visual appearance of the text; it does not set the selection.

setInsertionPosition: *position*
Set the *position* of the insert cursor. This message sets the insertion cursor position of the Text widget.

setSelection: *positions*
Set the selection. This message sets the primary selection of the text in the widget. It also sets the insertion cursor position to the last position of the selection.

setString: *value*
This message sets the string value of the Text widget.

setTopCharacter: *topCharacter*
This message sets the position of the text at the top of the Text widget. If the editMode is XmMULTILINEEDIT, the line of text that contains *topCharacter* is displayed at the top of the widget without shifting the text left or right.

showPosition: *position*
Force text at the specified position to be displayed. This message forces text at the specified position to be displayed.

tabSpacing: *anInteger*
Indicates the tab stop spacing.

topCharacter: *anInteger*
Displays the position of text at the top of the window. Position is determined by the number of characters from the beginning of the text.

value: *aString*
Specifies the displayed text String.

verifyBell: *aBoolean*
Specifies whether the bell should sound when the verification returns without continuing the action.

wordWrap: *aBoolean*
Indicates that lines are to be broken at word breaks (i.e., the text does not go off the right edge of the window).

Callbacks & Events

Activate Callback

These callbacks are triggered when the user presses the default action key. This is typically a carriage return.

Focus Callback

These callbacks are triggered before the widget has accepted input focus.

Losing Focus Callback

These callbacks are triggered before the widget loses input focus. This callback can be used to perform input validation of the user entered data.

Modify Verify Callback

These callbacks are triggered before text is deleted from or inserted into the widget. This callback can be used to check a character value after it is entered by the user and before it is accepted by the control.

Call data arguments:

text - a String which contains the text which is to be inserted.
currInsert - the current position of the insert cursor.
startPos - the starting position of the text to modify.
endPos - the ending position of the text to modify.

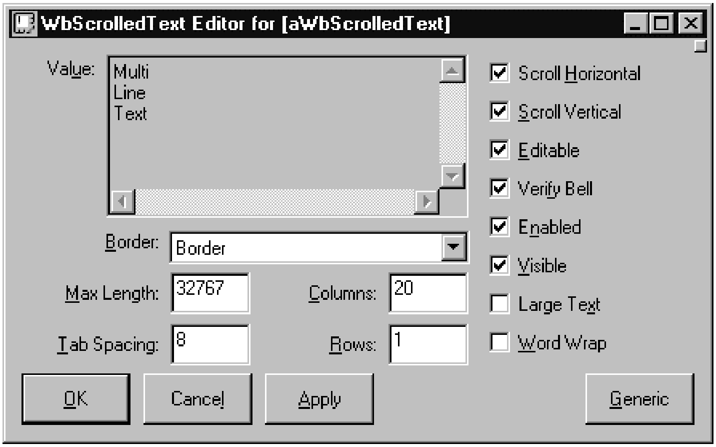
Resize Callback

These callbacks are triggered when the widget receives a resize event. This allows the widget to perform any calculation to adjust the size of the image that it displays.

Value Changed Callback

These callbacks are triggered after text is deleted from or inserted into the widget. This callback can be used to retrieve the current value of the widget.

Editor



WbScrolledText Editor for [aWbScrolledText]

Value: Multi
Line
Text

Border: Border

Max Length: 32767

Columns: 20

Tab Spacing: 8

Rows: 1

☑ Scroll Horizontal
☑ Scroll Vertical
☑ Editable
☑ Verify Bell
☑ Enabled
☑ Visible
☐ Large Text
☐ Word Wrap

Border Width

Specifies the width of the border that surrounds the widget's window on all four sides. The width is specified in pixels. A width of zero means that no border will show.

Border - Causes the widget to have a border.
No Border - Causes the widget to have no border.

Columns

Specifies the initial width of the text window measured in character spaces.

Editable

Indicates that the user can edit the text string when set to true. A false value will prohibit the user from editing the text.

Enabled

Determines whether a widget will react to input events. Disabled (insensitive) widgets do not react to input events.

Large Text

This is a hint that indicates that the receiver will be processing a large amount of text. If this flag is false, text operations may fail due to space limitations. If this hint is true, text operations will not fail.

Max Length

Specifies the maximum length of the text string that can be entered into text from the keyboard.

Rows

Specifies the initial height of the text window measured in character heights.

Scroll Horizontal

Adds a ScrollBar that allows the user to scroll horizontally through text.

Scroll Vertical

Adds a ScrollBar that allows the user to scroll vertically through text.

Tab Spacing

Indicates the tab stop spacing.

Value

Specifies the displayed text String.

Verify Bell

Specifies whether the bell should sound when the verification returns without continuing the action.

Visible

Maps the widget (makes visible) as soon as it is both realized and managed, if set to True. If set to False, the client is responsible for mapping and unmapping the widget.

Word Wrap

Indicates that lines are to be broken at word breaks (i.e., the text does not go off the right edge of the window).

Chapter 13 WbApplication Protocol

WindowBuilder Pro generates all window definitions as subclasses of WbApplication. WbApplication is a powerful and flexible abstract superclass providing a generalized windowing framework (which is not found in the base image). This chapter presents the public protocol supported by the WbApplication class. The following major protocol categories are covered:

- Opening and Closing
- Accessing
- Subclass Overrides
- Prompting
- Utility
- Mini Help Support
- Creating

Opening and Closing

close
Close the receiver's window.

closeAndExit
Close the window and shut down Smalltalk.

closeAndExitIfLast
Close the window and shut down Smalltalk if there are no other windows.

createWidget: *theName* **parent:** *parent* **argBlock:** *argBlock* **[class method]**
Create an instance of the receiver to be embedded in another application.

exitSystem
Shuts down Smalltalk

open
Create and realize the receiver.

open **[class method]**
Open an instance of the receiver.

openDialog
Create the receiver with a dialog shell as child of the current window. The dialog appears when the shell is managed.

openDialog: *aWidget*
Create the receiver with a dialog shell as child of *aWidget*. The dialog is application modal. The dialog appears when the shell is managed.

openDialog: *aWidget* **inputMode:** *inputMode*
Create the receiver with a dialog shell as child of *aWidget*. The input mode is specified via *inputMode*. The dialog appears when the shell is managed.

openDialog: *aWidget* **on:** anObject
Create the receiver with a dialog shell as child of *aWidget*. The dialog is application modal. The dialog appears when the shell is managed.

openDialog: *aWidget* **on:** *anObject* **inputMode:** *inputMode*
Create the receiver with a dialog shell as child of *aWidget*. An arbitrary object *anObject* may be passed in. The input mode is specified via *inputMode*. The dialog appears when the shell is managed.

openDialogModeless
Create the receiver with a dialog shell as child of the current window. The dialog appears when the shell is managed.

openDialogModeless: *aWidget*
Create the receiver with a dialog shell as child of *aWidget*. The dialog is modeless to its parent window. The dialog appears when the shell is managed.

openDialogOn: *anObject*
Create the receiver with a dialog shell as child of the current window. The dialog appears when the shell is managed.

openDialogParentModal
Create the receiver with a dialog shell as child of the current window. The dialog appears when the shell is managed.

openDialogParentModal: *aWidget*
Create the receiver with a dialog shell as child of *aWidget*. The dialog is modal to its parent window. The dialog appears when the shell is managed.

openDialogSystemModal
Create the receiver with a dialog shell as child of the current window. The dialog appears when the shell is managed.

openDialogSystemModal: *aWidget*
Create the receiver with a dialog shell as child of *aWidget*. The dialog is system modal. The dialog appears when the shell is managed.

openOn: *anObject*
Create and realize the receiver. An arbitrary object *anObject* may be passed in.

openWithParent: *aWbApplication*
Create and realize the receiver as a child of *aWbApplication*. The receiver will float above *aWbApplication*. Note that this is *not* the same as an MDI application.

Accessing

form
Answer the receiver's form.

form: *aWidget*
Set the receiver's form to *aWidget.*

mainWindow
Answer the receiver's main window.

mainWindow: *aWidget*
Set the receiver's main window to *aWidget.*

menuBar
Answer the receiver's menu bar.

menuBar: *aWidget*
Set the receiver's menu bar to *aWidget.*

menuNamed: *aString*
Answer the menu named *aString.*

parent
Answer the receiver's parent.

parent: *aWidget*
Set the receiver’s parent to *aWidget.*

properties
Answer the receiver’s property dictionary.

properties: *anIdentityDictionary*
Set the receiver’s property dictionary.

propertyAt: *aSymbol*
Answer the receiver’s property named *aSymbol*.

propertyAt: *aSymbol* **ifAbsent:** *aBlock*
Answer the receiver’s property named *aSymbol*. If no such property exists, evaluate *aBlock.*

propertyAt: *aSymbol* **ifAbsentPut:** *aBlock*
Answer the receiver’s property named *aSymbol*. If no such property exists, store the value of *aBlock* there.

propertyAt: *aSymbol* **ifMissing:** *anObject*
Answer the receiver’s property named *aSymbol*. If no such property exists, store *anObject* there.

propertyAt: *aSymbol* **put:** *anObject*
Set the receiver’s property named *aSymbol* to *anObject*.

screen
Answer the CgScreen for the receiver.

shell
Answer the receiver’s shell.

shell: *aWidget*
Set the receiver’s shell to *aWidget.*

topLevelShell
Answer the receiver’s topLevelShell widget.

widgetNamed: *aString*
Answer the child of the receiver’s form named *aString*.

Subclass Overrides

addApplicationMenus
Add the application menus to the menu bar.

addStandardLeftMenus
Add the standard left menus to the menu bar.

addStandardRightMenus
Add the standard right menus to the menu bar.

addSystemMenus
Add the system menus to the menu bar.

addWidgets
Add the widgets to the form.

closingWindow
This method is automatically invoked when closing a WbApplication window from the system menu. It returns a Boolean indicating whether the close should continue.

defaultFont
Answers the default font used by the application.

defaultTimerProc
The default procedure that is executed in response to a timer event.

destroyWindow
This method is automatically invoked when the receiver is destroyed.

dispidAmbientPropertyAt: *dispid*
Default ambient property handler for OLE/ActiveX widgets. Subclasses can override this method to customize behavior. Alternatively, consumers can set the `#ambientPropertyReceiver`: and `#ambientPropertySelector`: attributes of the OLE widget to provide a custom ambient property handler

embeddedFormClass
Return the default form class to use in embedding this application inside of another.

formClass
Return the default form class to use in the application.

initialize
Initialize the receiver.

initializeGraphics
Initialize any resources required for graphics drawing. Currently this is done before realizing the widget hierarchy so widget windows are not available.

initializeMenus
Initialize the menus with application specific behavior.

initializeShell
Initialize the shell.

initializeWidgets
Initialize the widgets with application specific behavior.

initialWindowPosition
Hook for dynamically setting the window’s position.

initialWindowSize
Hook for dynamically setting the window’s size.

initWindow
Initialize the application after it has been realized.

initWindow: *anObject*
Initialize the application with *anObject* after it has been realized.

mainWindowClass
Return the default main window class to use in the application.

preInitWindow
Initialize the application before it has been realized.

preInitWindow: *anObject*
Initialize the application with *anObject* before it has been realized.

setUpForm: *aForm*
Set up the create args for *aForm*.

setUpMainWindow: *aWindow*
Set up the create args for *aWindow*.

setUpShell: *aShell*
Set up the create args for *aShell*.

setUpShellCallbacks: *aShell*
Set up callbacks for *aShell*.

setUpWindowCallbacks
Set up the callbacks for the shell and main window.

shellClass
Return the default shell class to use in the application.

shellName
Answer the name of the receiver's shell. This is used as the default label of the window. The default is to answer the receiver's class name.

shouldAutoScale
Answers whether the window should auto scale based on the screen resolution. The default is true.

shouldAutoScaleUsingDefaultFont
Answer whether the system should auto scale using the default font returned by #defaultFont or the system font.

> true = use default font.
> false = use system font (this is the default mechanism)

shouldClearEventsOnExit
Answer whether the system should automatically process all remaining events when the window is closed.

windowTitle
Hook for dynamically setting the window's title.

Prompting

confirm: *aString*
Confirm the question posed by *aString* with the user. Answer true if the user chooses YES. Otherwise answer false.

confirmYesNoCancel: *aString*
Confirm the question posed by *aString* with the user using the most appropriate device or mechanism available. Answer true if the user chooses YES, false if the user chooses NO, otherwise answer nil.

message: *aString*
Inform the user of the information contained in *aString* using the most appropriate device or mechanism available. This method always answers nil.

message: *messageString* **title:** *titleString*
Inform the user of the information contained in messageString using the most appropriate device or mechanism available. This method always answers nil.

proceed: *aString*
Inform the user of the action described by *aString* that is about to take place and obtain a proceed/cancel response. Answer true if the user chooses OK. Otherwise answer false.

prompt: *message* **answer:** *suggestion*
Prompt the user for a typed response presenting the information contained in *message* as the prompt and *suggestion* as the default answer using the most appropriate device or mechanism available. This operation returns nil if the user canceled, or a String containing the user's response.

prompt: *message* **extendedSelectFrom:** *selectionList*
Open a list chooser and return the selected object from the choices. If the user cancels return nil.

prompt: *message* **extendedSelectFrom:** *selectionList* **selectedItems:** *selectedItems*
Open a list chooser and return the selected object from the choices. If the user cancels return nil. Set the initial selection to *selectedItems*.

prompt: *message* **multipleSelectFrom:** *selectionList*
Open a list chooser and return the selected object from the choices. If the user cancels return nil.

prompt: *message* **multipleSelectFrom:** *selectionList* **selectedItems:** *selectedItems*
Open a list chooser and return the selected object from the choices. If the user cancels return nil. Set the initial selection to *selectedItems.*

prompt: *message* **singleSelectFrom:** *selectionList*
Open a list chooser and return the selected object from the choices. If the user cancels return nil.

prompt: *message* **singleSelectFrom:** *selectionList* **dependentListBlock:** *aBlock*
Open a list chooser and return the selected object from the choices. If the user cancels return nil. The *aBlock* is a one parameter block that is evaluated with the selection from the first list to obtain the items for the dependent list when a selection is made in the first list.

prompt: *message* **singleSelectFrom:** *selectionList* **selectedItem:** *selectedItem*
Open a list chooser and return the selected object from the choices. If the user cancels return nil. Set the initial selection to *selectedItem.*

prompt: *message* **singleSelectFrom:** *selectionList* **selectedItem:** *selectedItem* **dependentListBlock:** *dependentListBlock*
Open a double list chooser and return the selected object from the choices. If the user cancels return nil. The *dependentListBlock* is a one parameter block that is evaluated with the selection from the first list to obtain the items for the dependent list when an selection is made in the first list.

prompt: *message* **title:** *title* **answer:** *suggestion*
Prompt the user for a typed response presenting the information contained in *message* as the prompt and *suggestion* as the default answer using the most appropriate device or mechanism available. This operation returns nil if the user canceled, or a String containing the user's response.

promptForFileName: *prompt*
Answer a file name entered by the user.

promptForFileName: *prompt* **defaultName:** *fileName*
Answer a file name entered by the user. Provide *fileName* as the default.

threeStateNotify: *title* **withText:** *aString*
Confirm the question posed by *aString* with the user using the most appropriate device or mechanism available. Answer true if the user chooses *yes*, false if the user chooses *no*, otherwise answer nil.

Utility

activate
Makes the receiver the active window and expands the receiver from an icon if necessary.

activeShell
Answer the current shell with focus.

addTimeout: *interval* **receiver:** *receiver* **selector:** *selector* **clientData:** *clientData*
This message allows a program to have a function called after a specified timeout. The message creates the timeout and returns an (opaque) identifier for it. The length of the timeout value is interval milliseconds. The specified callback is invoked when interval elapses, and the timeout is removed from the timeout queue. The return value is an object which uniquely identifies the pending timer pseudo-event. The pending event can be deleted from the queue before the interval expires by calling #removeTimeout:..

asyncExecFirstInUI: *aBlock*
Evaluate *aBlock* in the u/i Process. No result is returned. Processes with higher priority than the u/i will NOT block. In this case, *aBlock* is executed the next time the u/i becomes active. If this message is sent by the u/i process, then *aBlock* will be executed after all previously queued background graphic requests have been executed.

asyncExecInUI: *aBlock*
Evaluate *aBlock* in the u/i Process. No result is returned. Processes with higher priority than the u/i will NOT block. In this case, *aBlock* is executed the next time the u/i becomes active. If this message is sent by the u/i process, then *aBlock* will be executed after all previously queued background graphic requests have been executed.

allWindows **[class method]**
Answer a collection of all WbApplication windows that have realized shells.

backgroundColor: *aCgRGBColor*
Set the background color of the window and all non-text and non-list widgets to *aCgRGBColor*.

bell
Ring the system bell.

breakInLongOperation: *aBlock*
Process a system break message in a long operation.

checkMenuNamed: *aString*
Check the menu named *aString*. This only works with toggle menus.

clearEvents
Clear events until there are no more.

defaultFontExtent
Answer the size of the default font.

defaultGrayColor
Answer the default gray color.

defaultGrayValue
Answer the default gray value.

deferRedrawInShortOperation: *aBlock*
Defer redrawing the receiver during *aBlock*. Meanwhile, display the hour glass cursor.

disableMenuNamed: *aString*
Disable the menu named *aString.*

emptyCollection
This is a dummy method that can be used whenever a selector returning an OrderedCollection is required.

emptyIdentityDictionary
This is a dummy method that can be used whenever a selector returning an IdentityDictionary is required.

enableMenuNamed: *aString*
Enable the menu named *aString*.

execLongOperation: *aBlock*
Show a busy dialog, and fork a background process to evaluate *aBlock*, while keeping the user interface responsive. Return the result of evaluating *aBlock*. Show a busy cursor in the invoking window. The code in *aBlock* must obey the rules for non-UI processes i.e. direct UI operations are not permitted.

execLongOperation: *aBlock* **message:** *message*
Show a busy dialog, and fork a background process to evaluate *aBlock*, while keeping the user interface responsive. Return the result of evaluating *aBlock*. Show a busy cursor in all WbApplication windows. The code in *aBlock* must obey the rules for non-UI processes i.e. direct UI operations are not permitted.

execLongOperation: *aBlock* **message:** *message* **allowCancel:** *allowCancel* **showProgress:** *showProgress*
Show a busy dialog, and fork a background process to evaluate aBlock, while keeping the user interface responsive. Return the result of evaluating aBlock. Show a busy cursor in all WbApplication windows. The code in aBlock must obey the rules for non-UI processes i.e. direct UI operations are not permitted. If allowCancel is set to true, a cancel button will be displayed. The application can determine if the operation has been cancelled by sending #cancelled to the dialog periodically. If showProgress is true, percentage complete will be shown. The application must update the progress by sending #fractionComplete: to the dialog with a fraction between 0 and 1. If the block defines a parameter, the dialog is passed as the block parameter.

execLongOperation: *aBlock* **message:** *message* **errorBlock:** *errorBlock*
Show a busy dialog, and fork a background process to evaluate *aBlock*, while keeping the user interface responsive. Return the result of evaluating *aBlock*. Show a busy cursor in all WbApplication windows. The code in *aBlock* must obey the rules for non-UI processes i.e. direct UI operations are not permitted.

execLongOperation: *aBlock* **message:** *message* **title:** *title*
Show a busy dialog, and fork a background process to evaluate *aBlock*, while keeping the user interface responsive. Return the result of evaluating *aBlock*. Show a busy cursor in all WbApplication windows. The code in *aBlock* must obey the rules for non-UI processes i.e. direct UI operations are not permitted.

execLongOperation: *aBlock* **message:** *message* **title:** *title* **allowCancel:** *allowCancel* **showProgress:** *showProgress*

Show a busy dialog, and fork a background process to evaluate aBlock, while keeping the user interface responsive. Return the result of evaluating aBlock. Show a busy cursor in all WbApplication windows. The code in aBlock must obey the rules for non-UI processes i.e. direct UI operations are not permitted. If allowCancel is set to true, a cancel button will be displayed. The application can determine if the operation has been cancelled by sending #cancelled to the dialog periodically. If showProgress is true, percentage complete will be shown. The application must update the progress by sending #fractionComplete: to the dialog with a fraction between 0 and 1. If the block defines a parameter, the dialog is passed as the block parameter.

execLongOperation: *aBlock* **message:** *message* **title:** *title* **allowCancel:** *allowCancel* **showProgress:** *showProgress* **errorBlock:** *errorBlock*
Show a busy dialog, and fork a background process to evaluate aBlock, while keeping the user interface responsive. Return the result of evaluating aBlock. Show a busy cursor in all WbApplication windows. The code in aBlock must obey the rules for non-UI processes i.e. direct UI operations are not permitted. If allowCancel is set to true, a cancel button will be displayed. The application can determine if the operation has been cancelled by sending #cancelled to the dialog periodically. If showProgress is true, percentage complete will be shown. The application must update the progress by sending #fractionComplete: to the dialog with a fraction between 0 and 1. If the block defines a parameter, the dialog is passed as the block parameter.

execLongOperation: *aBlock* **message:** *message* **title:** *title* **errorBlock:** *errorBlock*
Show a busy dialog, and fork a background process to evaluate *aBlock*, while keeping the user interface responsive. Return the result of evaluating *aBlock*. Show a busy cursor in all WbApplication windows. The code in *aBlock* must obey the rules for non-UI processes i.e. direct UI operations are not permitted.

execShortOperation: *aBlock*
Evaluate *aBlock* in the UI process while showing a busy cursor in the active shell.

false
This is a dummy boolean method that can be used whenever a boolean selector method is required.

flushEvents
Dispatch events until there are no more.

foregroundColor: *aCgRGBColor*
Set the foreground color of the window and all child widgets to *aCgRGBColor.*

inProgressDialog
Answer the in-progress dialog

inProgressDialog: *aShell*
Set the in-progress dialog to *aShell*.

isAltKeyDown
Answer whether the Alt key is down.

isControlKeyDown
Answer whether the Control key is down.

isShiftKeyDown
Answer whether the Shift key is down.

nil
This is a dummy method that can be used whenever a selector returning nil is required.

onCharacter: *aCharacter* **do:** *aBlock*
When Alt + *aCharacter* is hit, evaluate *aBlock*.

onCharacter: *aCharacter* **perform:** *aSymbol*
When Alt + *aCharacter* is hit, perform *aSymbol*.

onCharacter: *aCharacter* **perform:** *aSymbol* **with:** *anObject*
When Alt + *aCharacter* is hit, perform *aSymbol* with *anObject* as an argument.

onCharacter: *aCharacter* **perform:** *aSymbol* **with:** *anObject* **with:** *anObject2*
When Alt + *aCharacter* is hit, perform *aSymbol* with *anObject* and *anObject2* as arguments.

onCharacter: *aCharacter* **perform:** *aSymbol* **with:** *anObject* **with:** *anObject2* **with:** *anObject3*
When Alt + *aCharacter* is hit, perform *aSymbol* with *anObject*, *anObject2* and *anObject3* as arguments.

onCharacter: *aCharacter* **perform:** *aSymbol* **withArguments:** *anArray*
When Alt + *aCharacter* is hit, perform *aSymbol* with *anArray* as arguments.

onCharacter: *aCharacter* **send:** *aDirectedMessage*
When Alt + *aCharacter* is hit, send *aDirectedMessage*.

onControlChar: *aCharacter* **do:** *aBlock*
When Control + *aCharacter* is hit, evaluate *aBlock*.

onControlChar: *aCharacter* **perform:** *aSymbol*
When Control + *aCharacter* is hit, perform *aSymbol*.

onControlChar: *aCharacter* **perform:** *aSymbol* **with:** *anObject*
When Control + *aCharacter* is hit, perform *aSymbol* with *anObject* as an argument.

onControlChar: *aCharacter* **perform:** *aSymbol* **with:** *anObject* **with:** *anObject2*
When Control + *aCharacter* is hit, perform *aSymbol* with *anObject* and *anObject2* as arguments.

onControlChar: *aCharacter* **perform:** *aSymbol* **with:** *anObject* **with:** *anObject2* **with:** *anObject3*
When Control + *aCharacter* is hit, perform *aSymbol* with *anObject*, *anObject2* and *anObject3* as arguments.

onControlChar: *aCharacter* **perform:** *aSymbol* **withArguments:** *anArray*
When Control + *aCharacter* is hit, perform *aSymbol* with *anArray* as arguments.

onControlChar: *aCharacter* **send:** *aDirectedMessage*
When Control + *aCharacter* is hit, send *aDirectedMessage*.

removeTimeout: *identifier*
This message removes the timeout specified by identifier. Identifier is the value returned by a #addTimeout: call. Note that timeouts are automatically removed once they expire and the callback has been called. The identifier is the Association returned by the addTimeout: call. The timerProcs are a SortedCollection.

setGrayBackgroundColor
Set the window background color gray.

setMenuNamed: *aString* **labelString:** *newLabel*
Set the menu named *aString* label string to *newLabel*.

showBusyCursorInAllWindowsWhile: *aBlock* **[class method]**
Show a busy cursor in all WbApplication windows while evaluating *aBlock*.

startTimer: *anIntegerOrSymbol* **period:** *milliseconds*
Start a timer identified by *anIntegerOrSymbol* with a period of *milliseconds*. If *anIntegerOrSymbol* is a symbol, the method identified by the symbol will be executed. If *anIntegerOrSymbol* is an integer, the #defaultTimerProc method will be executed. The #timer callback will also be triggered for each timer event.

stopAllTimers
Stop all timers associated with the window.

stopTimer: *anIntegerOrSymbol*
Stop the timer identified by *anIntegerOrSymbol*.

suspendExecutionUntilRemoved
If the shell is not a dialog block until the shell is destroyed. If the shell is a dialog, the widget does not get destroyed until the parent gets destroyed but gets unmanaged when it is closed. Therefore if it is a dialog don't continue once it is unmanaged.

true
This is a dummy boolean method that can be used whenever a boolean selector method is required.

uncheckMenuNamed: *aString*
Uncheck the menu named *aString*. This only works with toggle menus.

Mini Help Support

initializeMiniHelp
Turns on mini help for your window (call this in your `#preInitWindow` method. The mini help text selector defaults to `#miniHelpTextFor:`.

initializeMiniHelp: *aSelector*
Turns on mini help and allows you to set an alternative mini help text selector.

initializeMiniHelpIfNecessary
Initializes mini help if it has not already been initialized.

miniHelpBackColor
Answers the background color to be used for the mini help window.

miniHelpClear
Popdown the Mini Help window.

miniHelpDelay
Answers the delay in milliseconds between the cursor entering the bounds of a widget and the mini help appearing.

miniHelpDelay: *millisecondCount*
Sets the delay in milliseconds between the cursor entering the bounds of a widget and the mini help appearing.

miniHelpEnabled
Answers whether mini help is currently enabled or not.

miniHelpEnabled: *aBoolean*
Sets whether mini help is currently enabled or not.

miniHelpFont
Answers the font to be used in the mini help window.

miniHelpForeColor
Answers the foreground color to be used for the mini help window.

miniHelpSelector
Answers the mini help selector.

miniHelpSelector: *aSymbol*
Sets the mini help selector.

miniHelpTextFor: *aWidget*
Answers the help text for the widget pass in as the argument.

showMiniHelpFor: *aWidget*
Pop up the mini help window for *aWidget*.

tipText: *aString*
Set the tip text for the receiver.

Creating

buildAcceleratorTable
Build the accelerator table. This provides emulation for button keyboard accelerators.

createForm
Add a form to the main window.

createMainWindow
Add a main window to the shell.

createMenuBar
Create the receiver’s menuBar widget.

createMenus
Create the menus.

createPulldownMenus
Create the pulldown menus.

createShell
Create the shell.

createWindow
Create the window

createWorkRegion
Create the work region

destroyMenuBar
Destroy the menu bar.

do: *aBlock* **withForm:** *aForm*
Execute *aBlock* with the application’s form set to *aForm*.

do: *aBlock* **withMenuBar:** *aMenuBar*
Execute *aBlock* with the application’s form set to *aMenuBar*.

formArgs
Answer the arg block used by the form.

mainWindowArgs
Answer the arg block used by the main window.

manageShell
Manage the shell.

manageWidgets
Manage the widgets.

manageWidgetsFor: *aForm*
Manage the widgets for *aForm*.

mapShell
Map the shell.

menuBarArgs
Answer the create args for the menu bar.

newMenu
Return a new, empty menu.

realizeWindow
Realize the receiver’s widget hierarchy.

setInputMode: *inputMode*
Set the input mode for a modal dialog.

setMainWindowAreas
Set the main window areas. The default is to have a menu bar and a work region and no main scroll bars.

setUpMainWindowCallbacks: *aWindow*
Set up callbacks for *aWindow*.

shellArgs
Answer the arg block used by the shell.

Appendix A Customizing WindowBuilder Pro

WindowBuilder Pro is highly customizable and may be easily tailored with additional capabilities that go beyond those available “out of the box”. This appendix discusses the four major customization opportunities that are available to the developer:

- Adding support for new widgets
- Building a custom attribute editor
- Using Add-In modules
- Adding code generation

Adding Support for New Widgets

WindowBuilder Pro manipulates real instances of widgets in the main editor window. In order to properly manipulate a widget, WindowBuilder Pro needs to know a few things about it. Whenever a widget type is used for the first time, an instance of a WbAttributeManager is created for it. There will be one instance of WbAttributeManager for every widget in the system. This attribute manager is used for things like code generation, attribute editing, copying, morphing, etc. Without intervention on the part of the developer, the system will build a default attribute manager based on information provided by the widget’s superclasses. Attributes that are local to the widget itself will not be included without some additional effort. WindowBuilder Pro provides a number of protocols in the CwWidget class that may be overridden in subclasses to customize their attributes.

The available protocols are:

- wbAbstractClass
- wbAttributeEditorClass
- wbAttributeList
- wbAttributeComments
- wbAttributeResourceValues

- wbCallbackNames
- wbCallbackAnnotations
- wbCallbackDefaultDataSelectors
- wbCanTab
- wbCloneSpecialAttributesFrom
- wbColorAttributes
- wbDefaultParentScrollingPolicy
- wbDefaultValues
- wbDirectEditManager
- wbFontAttributeList
- wbImportantCallbacks
- wbImportantMessages
- wbIsomorphicClasses
- wbMaxSize
- wbMinSize
- wbOverrideDefaultAttributes:
- wbProcessEditorEvent:
- wbRequiredPoolDictionaries
- wbStyleAttribute
- wbSubStyleAttribute

wbAbstractClass
Answer whether the widget class is abstract and cannot be placed within WindowBuilder Pro. This method must return false for any concrete widget class. If the widget is subclassed from an existing concrete class, this method is already set up for you.

wbAttributeEditorClass
Answer the class to use as an attribute editor. See the following section, *Building a Custom Attribute Editor*, for details. The following is an example from CwList:

```
wbAttributeEditorClass
   ^WbScrolledListEditor
```

wbAttributeList

Answer a list of valid attributes for the widget. This method should call `super wbAttributeList` and then add a collection of symbols representing any non-inherited attributes of the widget. Each attribute that is specified should have a corresponding get and set method with the same name. The following is an example from CwList:

```
wbAttributeList
   ^super wbAttributeList
      addAll: #(
         selectionPolicy
         browseSelectionCallback
         defaultActionCallback
         extendedSelectionCallback
         items
         multipleSelectionCallback
         scrollHorizontal
         selectedItems
         singleSelectionCallback
         visibleItemCount
         topItemPosition
         );
      yourself.
```

wbAttributeComments

Answer a dictionary where the keys are attribute symbols and the values are the descriptions of that attribute. These attribute comments are used to describe the attribute in the generic attribute editor and the callback editor as well as in balloon help in the custom attribute editors. The following is an example from CwList:

```
wbAttributeComments
   ^super wbAttributeComments
      at: #defaultActionCallback put: ‘These callbacks are
         triggered when an item is double clicked.’;
      at: #items put: ‘An array of Strings that are to be
         displayed as the list items.’;
      at: #selectedItems put: ‘An array of Strings that
         represents the list items that are currently
         selected, either by the user or the application.’;
      at: #selectionPolicy put: ‘Defines the interpretation
         of the selection action.’;
      at: #singleSelectionCallback put: ‘These callbacks
         are triggered when an item is selected in single
         selection mode. It is only valid when Selection
         Policy is Single Select.’;
      yourself.
```

wbAttributeResourceValues

Answer a dictionary where the keys are attribute symbols and the values are arrays containing a list of WbResourceDescriptors representing valid keys in the CwConstants or EwConstants pool dictionary. The first item in the array should be the default value. The following is an example from CwList:

```
wbAttributeResourceValues
   ^super wbAttributeResourceValues
      at: #selectionPolicy put: (OrderedCollection new
         add: (WbResourceDescriptor
            name: ‘XmSINGLESELECT’
            commonName: ‘Single Select’
            comment: ‘Allows only single selections. Under
               Windows and OS/2, this is the same as
               Browse Select’);
         add: (WbResourceDescriptor
            name: ‘XmMULTIPLESELECT’
            commonName: ‘Multiple Select’
            comment: ‘Allows multiple items to be
               selected. The selection of an item is
               toggled when it is clicked on. Clicking on
               an item does not deselect previously
               selected items.’);
         add: (WbResourceDescriptor
            name: ‘XmEXTENDEDSELECT’
            commonName: ‘Extended Select’
            comment: ‘Allows multiple items to be
               selected, either by dragging the selection
               or by clicking on items with a modifier key
               held down. Clicking on an item without a
               modifier key held down deselects all
               previously selected items.’);
         add: (WbResourceDescriptor
            name: ‘XmBROWSESELECT’
            commonName: ‘Browse Select’
            comment: ‘Allows only single selection. The
               selection changes when the mouse is
               dragged. This is the default Selection
               Policy. Under Windows and OS/2, this is the
               same as Single Select’);
         yourself);
      yourself.
```

wbCallbackNames
Answer a dictionary where the keys are callback attribute symbols and the values are the names of that attribute. This method maps callback attributes to their corresponding callback names. This allows WindowBuilder Pro to use the correct callback name when it generates code. The following is an example from CwList:

```
wbCallbackNames
   ^super wbCallbackNames
      at: #defaultActionCallback put:
         ‘XmNdefaultActionCallback’;
      at: #extendedSelectionCallback put:
         ‘XmNextendedSelectionCallback’;
      at: #multipleSelectionCallback put:
         ‘XmNmultipleSelectionCallback’;
      at: #singleSelectionCallback put:
         ‘XmNsingleSelectionCallback’;
      yourself.
```

wbCallbackAnnotations
Answer a dictionary where the keys are callback attribute symbols and the values are the annotations providing further information about that attribute (e.g., explanations of the callData). These annotations are used in the generated callback stubs when “Use Long Callback Comments” is enabled. The following is an example from CwList:

```
wbCallbackAnnotations
   ^super wbCallbackAnnotations
     at: #defaultActionCallback put: #(
        ‘item - the String which is the selected item.’
        ‘itemPosition - the integer position of the
         selected item in the list.’
        );
     at: #singleSelectionCallback put: #(
        ‘item - the String which is the selected item.’
        ‘itemPosition - the integer position of the
         selected item in the list.’
        );
     at: #multipleSelectionCallback put: #(
        ‘item - the String which is the selected item.’
        ‘itemPosition - the integer position of the
         selected item in the list.’
        ‘selectedItemCount - the integer number of
         selected items.’
        ‘selectedItemPositions - a Collection containing
         the integer positions of the selected items.’
        ‘selectedItems - a Collection of Strings which are
         the selected items.’
        );
     yourself.
```

wbCallbackDefaultDataSelectors
Answer a dictionary where the keys are callback attribute symbols and the values are the selectors retrieving the default data for the callback. The following is an example from CwText:

```
wbCallbackDefaultDataSelectors
   ^super wbCallbackDefaultDataSelectors
      at: #modifyVerifyCallback put: #value;
      at: #valueChangedCallback put: #value;
      yourself.
```

wbCanTab
Answer whether the receiver can be in the tab order. The default is true. The following is an example from CwLabel:

```
wbCanTab
   ^false
```

wbCloneSpecialAttributesFrom: *aCwWidget*
Clone special attributes of *aCwWidget* into the receiver. In some cases, the base system does not cleanly handle all attributes when used in copying and morphing operations (this can be the case when an attribute is maintained by the operating system and not by an instance variable of the widget itself). In those cases, it is necessary to set the *useInCloning* flag of the attribute to false and handle the attribute copying manually in this method. The following is an example from CwList:

```
wbCloneSpecialAttributesFrom: aCwWidget
   super wbCloneSpecialAttributesFrom: aCwWidget.
   aCwWidget wbItems notNil
      ifTrue: [self wbItems: aCwWidget wbItems copy].
```

wbColorAttributes
Answer the attributes to be used for the color editor. This method would only be overridden for widget types that supported more than just the two default color attributes. The following is an example from CwPrimitive:

```
wbColorAttributes
   ^#(
      backgroundColor
      foregroundColor
      ).
```

wbDirectEditManager
Answer a suitable direct edit manager. WindowBuilder Pro allows the user to direct edit a widget by ALT-clicking on it. This method allows you to customize the type of direct editor that is invoked for the widget at hand. Several examples follow:

For CwList:

```
wbDirectEditManager
   ^WbDirectEditManager new
      owner: self extendedWidgetOrSelf;
      type: #multiLine;
      get: #items;
      set: #items:;
      filter: [:string | string wbArrayOfLines];
      yourself
```

For CwPushButton:

```
wbDirectEditManager
   ^WbDirectEditManager new
      owner: self extendedWidgetOrSelf;
      type: #singleLineInset;
      get: #labelString;
      set: #labelString:;
      filter: nil;
      yourself
```

For CwLabel:

```
wbDirectEditManager
   ^WbDirectEditManager new
      owner: self extendedWidgetOrSelf;
      type: #multiLine;
      get: #labelString;
      set: #labelString:;
      filter: nil;
      yourself
```

Four aspects of the direct editor may be customized:

- **type** - specifies the type of editor to pop up. Choices are `#multiLine`, `#singleLine` and `#singleLineInset`.
- **get** - specifies the get selector used to retrieve the value from the widget for initially setting the contents of the direct editor.
- **set** - specifies the set selector used for updating the widget once direct editing is finished.
- **filter** - specifies a filter (a block) to be applied to the direct editor’s result string before it is passed to the widget. The direct editor only deals with strings. Some widgets, like lists, want an array of strings for their contents. Using the block “`[:string | string wbArrayOfLines]`” will convert a multi-line string into an array of strings (one for each line).

wbDefaultParentScrollingPolicy
Answer the default scrolling policy when the receiver is placed within a CwScrolledWindow. This method can either return XmAUTOMATIC or XmAPPLICATIONDEFINED.

wbDefaultValues
Answer a dictionary where the keys are attribute symbols and the values are the default values of that attribute. Generally, the default values for each attribute will be determined automatically by the system when it first builds the attribute manager. This method need only be used to provide defaults for attributes that would otherwise register as nil or to override attributes that may return the wrong default value. The following is an example from CwList:

```
wbDefaultValues
   ^super wbDefaultValues
      at: #items put: OrderedCollection new;
      at: #selectedItems put: OrderedCollection new;
      at: #font put: CgDisplay default defaultFontStruct
         name asPortableFontString;
      yourself.
```

wbFontAttributeList
Answer a list of valid font attributes for the widget. It provides a hint to the system to use the standard font selection dialog when dealing with these attributes. The following is an example from CwList:

```
wbFontAttributeList
   ^super wbFontAttributeList
      addAll: #(
         font
         );
      yourself.
```

wbImportantCallbacks
Answer a list of important callbacks for the widget. These are the items that show up first in the callback list in the Callback Editor and in the popup Connect menu. The following is an example from CwPushButton:

```
wbImportantCallbacks
   ^super wbImportantCallbacks
      addFirst: #activateCallback;
      yourself
```

wbImportantMessages
Answer a list of important messages for the widget. These are the items that show up when performing a visual connection to the widget. The following is an example from CwToggleButton:

```
wbImportantMessages
   ^super wbImportantMessages
      addAll: #(
         #click
         #check
         #turnOff
         #turnOn
         #uncheck
         #set:
         );
      yourself
```

wbIsomorphicClasses
Answer the classes that the receiver can be morphed to. The following is an example from CwLabel:

```
wbIsomorphicClasses
   ^#(
      #CwLabel
      #CwText
      #WbEnhancedText
      #CwDrawnButton
      #CwPushButton
      #CwToggleButton
      )
```

wbMaxSize
Answer the maximum size of the widget. This method would only be overridden if the widget needed to constrain its size in one direction or another. The following is an example from CwText:

```
wbMaxSize
   self isDestroyed ifTrue: [^super wbMaxSize].
   ^super wbMaxSize x @ self preferredExtent y.
```

wbMinSize
Answer the minimum size of the widget. This method would only be overridden if the widget needed to constrain its size in one direction or another. The following is an example from CwText:

```
wbMinSize
   self isDestroyed ifTrue: [^0@0].
   ^super wbMinSize x @ self preferredExtent y.
```

wbOverrideDefaultAttributes: *attributeManager*
Give the class the opportunity to override the default attribute manager set up for the widget. Individual attributes may be augmented with additional flag information or default characteristics may be changed by this method. This method is used to supply information that is not captured by any of the other methods. The following is an example from CwList:

```
wbOverrideDefaultAttributes: attributeManager
   super wbOverrideDefaultAttributes: attributeManager.
   (attributeManager attributes at: #x)
         useParentIfScrolled: true.
   (attributeManager attributes at: #y)
         useParentIfScrolled: true.
   (attributeManager attributes at: #topItemPosition)
         isEditable: false.
   (attributeManager attributes at: #selectionPolicy)
         userDefault: XmSINGLESELECT.
   (attributeManager attributes at: #items)
      location: nil;
      getSelector: #wbItems;
      useInCloning: false;
      yourself.
```

A number of fields and flags may be set for each attribute:

- **default:** *anObject* - modify the (system) default value. This value is used to determine whether the attribute should be emitted during code generation. If the value of the attribute is the same as the default value, no code will be generated for that attribute.
- **getSelector:** *aSymbol* - modify the get selector.
- **isEditable:** *aBoolean* - is the attribute editable or not? If it is not editable, it will not appear in the generic attribute editor.
- **isPixmapAttribute:** *aBoolean* - is the attribute a pixmap attribute? This is a signal to use the graphics editor when editing this attribute.
- **location:** *anIntegerOrNil* - set the location to nil. The location specifies whether an attribute is a direct instance variable of the widget. This information is used to optimize cloning and morphing operations.
- **setSelector:** *aSymbol* - modify the set selector. The set selector is used in code generation.
- **useInCloning:** *aBoolean* - should the attribute be used when a widget is copied?
- **useInCodeGeneration:** *aBoolean* - should the attribute be used in code generation?
- **useInMorphing:** *aBoolean* - should the attribute be used in morphing?

- **useParentIfScrolled:** *aBoolean* - should this attribute use the parent widget's value when the widget is within a scrolled window? This is generally only used for the x, y, width and height attributes of scrolled lists and scrolled texts.
- **userDefault:** *anObject* - modify the user default. These defaults can also be set by the user using the Template Editor. The user default is used in initially instantiating a widget. Note that this value may be different from the system default value.

wbProcessEditorEvent: *event*
Process an event in the editor. Normally the event's point will be extracted and used for further processing. Answer false to indicate that the event was not handled. See EwPMNotebook and EwWINNotebook for examples.

wbRequiredPoolDictionaries
Answer the list of required pool dictionaries required by any instance of the receiver. This method would be overridden only in the case that a widget's code generation would reference constants in a pool dictionary other than CwConstants. Any pool dictionaries specified by this method would be automatically appended to the WbApplication definition in which the widget was added. The following is an example from EwNotebook:

```
wbRequiredPoolDictionaries
   ^#(EwConstants CwConstants)
```

wbStyleAttribute
Answer the attribute to be used for the style combobox. This provides a fast path to the single most important style attribute of a widget. The following is an example from CwList:

```
wbStyleAttribute
   ^#selectionPolicy
```

wbSubStyleAttribute
Answer the attribute to be used for the sub-style combobox. This provides a fast path to the second most important style attribute of a widget. The following is an example from CwComboBox:

```
wbSubStyleAttribute
   ^#editable
```

You should have the "WindowBuilder Pro – Tools" configuration loaded whenever you are adding enhancements or extending the definitions of any widgets. WindowBuilder Pro caches all widget attributes (for performance reasons), so it is important to re-initialize the widget attribute cache after you make any changes that affect widget attributes (like defining new callbacks, code generation attributes, WbEnhancedText validations, etc.). You can re-initialize the widget cache by executing the "Tools | Initialize | Widgets" command in the WindowBuilder Pro menubar (available when you load the WindowBuilder Pro Tools config. Alternatively, you can execute "WbAttributeManager initializeWidgets" in a workspace

Building a Custom Attribute Editor

WindowBuilder Pro provides a framework for building custom attribute editors. The class WbAbstractAttributeEditor is the superclass of all attribute editors. It does all of the work of matching up attribute values from a selected widget to fields in the custom attribute editor.

The steps involved to build a new editor are simple:

1. Start with an empty window.
2. For each attribute you wish to make available for editing, add a field to the window. String and Integer values should be placed as CwText or WbScrolledText. Boolean values should be CwToggleButtons. Attributes with multiple resource values defined should be CwComboBoxes. Attributes represented by arrays of strings should us a WbListItemEditor. WbListItemEditor is a WbApplication subclass and may be added to a window via holding the ALT key down while invoking the "Add Nested Application..." command.
3. As each field is added to the screen, its name should be set to exactly match the name of its corresponding attribute. The name is used by WbAbstractAttributeEditor to match the fields with the correct values.
4. Add CwLabels to any CwText and CwComboBox fields. There names should be "<attribute name>Label".
5. Add four buttons to the window: OK, Cancel, Apply and Generic.
6. Add the following callback handlers to the Activate Callback of each button:
    - OK - `#ok:clientData:calldata:`
    - Cancel - `#cancel:clientData:calldata:`
    - Apply - `#apply:clientData:calldata:`
    - Generic - `#generic:clientData:calldata:`

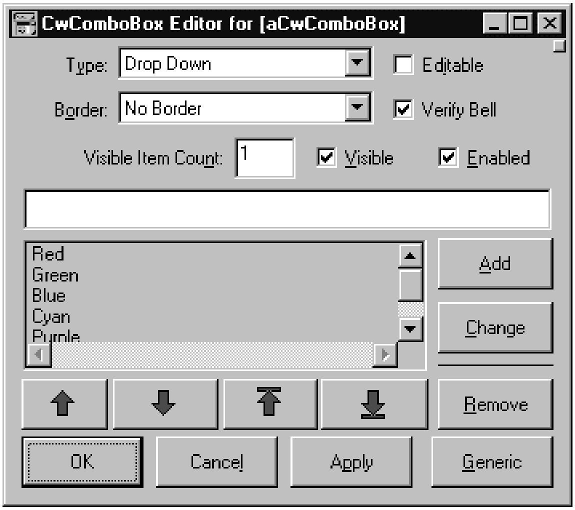
Note that each of these handlers is inherited from WbAbstractAttributeEditor and need not be overridden.

7. If a widget requires a pixmap attribute (e.g., its *isPixmapAttribute* flag is set to true), add the field as a CwPushButton. The button's Activate Callback should be `#pixmap:clientData:callData:`.
8. Use the Tab & Z-Order Editor to set the tab order.
9. Use the Attachment Editor to specify attachments.

10. Save the window. Hold the ALT key down when invoking the save dialog. You can then select WbAbstractAttributeEditor as the superclass. Make sure that the application in which you save the window has the WindowBuilder Pro application as prerequisite.

11. Add a `#wbAttributeEditorClass` instance method to the widget that returns the WbAbstractAttributeEditor that you just created.

As an example of the above, examine the WbComboBoxEditor class in WindowBuilder Pro:



CwComboBox Editor for [aCwComboBox]

Type: Drop Down

Border: No Border

Visible Item Count: 1

☐ Editable

☑ Verify Bell

☑ Visible

☑ Enabled

Red
Green
Blue
Cyan
Purple

The `#addWidgets` method for the above window is the following:

```
addWidgets
   “Private: WARNING!!!! This method was
    automatically generated by WindowBuilder Pro.
    Code you add here which does not conform to
    the WindowBuilder Pro API will probably be lost
    the next time you save your layout definition.”

   | apply borderWidth borderWidthLabel cancel
     comboBoxType comboBoxTypeLabel editable
     enabled generic items ok verifyBell visible
     visibleItemCount visibleItemCountLabel |
comboBoxTypeLabel := CwLabel
   createWidget: ‘comboBoxTypeLabel’
   parent: self form
   argBlock: [:w | w
      x: 4;
      y: 4;
      width: 60;
      height: 24;
      alignment: XmALIGNMENTEND;
      labelString: ‘Type:’;
      scale].
comboBoxType := CwComboBox
   createWidget: ‘comboBoxType’
   parent: self form
   argBlock: [:w | w
      x: 68;
      y: 4;
      width: 160;
      height: 24;
      scale].
borderWidthLabel := CwLabel
   createWidget: ‘borderWidthLabel’
   parent: self form
   argBlock: [:w | w
      x: 4;
      y: 32;
      width: 60;
      height: 24;
      alignment: XmALIGNMENTEND;
      labelString: ‘Border:’;
      scale].
borderWidth := CwComboBox
   createWidget: ‘borderWidth’
   parent: self form
   argBlock: [:w | w
      x: 68;
      y: 32;
      width: 160;
      height: 24;
      scale].
```

```
editable := CwToggleButton
   createWidget: ‘editable’
   parent: self form
   argBlock: [:w | w
      x: 240;
      y: 4;
      width: 76;
      height: 24;
      navigationType: XmTABGROUP;
      labelString: ‘Editable’;
      scale].
verifyBell := CwToggleButton
   createWidget: ‘verifyBell’
   parent: self form
   argBlock: [:w | w
      x: 240;
      y: 32;
      width: 92;
      height: 24;
      navigationType: XmTABGROUP;
      labelString: ‘Verify Bell’;
      scale].
visibleItemCountLabel := CwLabel
   createWidget: ‘visibleItemCountLabel’
   parent: self form
   argBlock: [:w | w
      x: 4;
      y: 60;
      width: 132;
      height: 28;
      alignment: XmALIGNMENTEND;
      labelString: ‘Visible Item Count:’;
      scale].
visibleItemCount := CwText
   createWidget: ‘visibleItemCount’
   parent: self form
   argBlock: [:w | w
      x: 140;
      y: 60;
      width: 40;
      height: 28;
      largeText: false;
      scale].
```

```
visible := CwToggleButton
   createWidget: ‘visible’
   parent: self form
   argBlock: [:w | w
      x: 192;
      y: 60;
      width: 68;
      height: 28;
      navigationType: XmTABGROUP;
      labelString: ‘Visible’;
      scale].
enabled := CwToggleButton
   createWidget: ‘enabled’
   parent: self form
   argBlock: [:w | w
      x: 268;
      y: 60;
      width: 76;
      height: 28;
      navigationType: XmTABGROUP;
      labelString: ‘Enabled’;
      scale].
items := WbListItemEditor
   createWidget: ‘items’
   parent: self form
   argBlock: [:w | w
      x: 8;
      y: 92;
      width: 336;
      height: 154;
      items: #();
      scale].
ok := CwPushButton
   createWidget: ‘ok’
   parent: self form
   argBlock: [:w | w
      x: 8;
      y: 248;
      width: 76;
      height: 32;
      navigationType: XmTABGROUP;
      labelString: ‘OK’;
      showAsDefault: 1;
      scale].
```

```
cancel := CwPushButton
   createWidget: ‘cancel’
   parent: self form
   argBlock: [:w | w
      x: 92;
      y: 248;
      width: 76;
      height: 32;
      navigationType: XmTABGROUP;
      labelString: ‘Cancel’;
      scale].
apply := CwPushButton
   createWidget: ‘apply’
   parent: self form
   argBlock: [:w | w
      x: 176;
      y: 248;
      width: 76;
      height: 32;
      navigationType: XmTABGROUP;
      labelString: ‘Apply’;
      scale].
generic := CwPushButton
   createWidget: ‘generic’
   parent: self form
   argBlock: [:w | w
      x: 264;
      y: 248;
      width: 76;
      height: 32;
      navigationType: XmTABGROUP;
      labelString: ‘Generic’;
      scale].
comboBoxTypeLabel
   attachLeft: 4 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   yourself.
comboBoxType
   attachLeft: 68 relativeTo: XmATTACHFORM;
   attachRight: 120 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   yourself.
borderWidthLabel
   attachLeft: 4 relativeTo: XmATTACHFORM;
   attachTop: 32 relativeTo: XmATTACHFORM;
   yourself.
```

```
borderWidth
   attachLeft: 68 relativeTo: XmATTACHFORM;
   attachRight: 120 relativeTo: XmATTACHFORM;
   attachTop: 32 relativeTo: XmATTACHFORM;
   yourself.
editable
   attachRight: 32 relativeTo: XmATTACHFORM;
   attachTop: 4 relativeTo: XmATTACHFORM;
   yourself.
verifyBell
   attachRight: 16 relativeTo: XmATTACHFORM;
   attachTop: 32 relativeTo: XmATTACHFORM;
   yourself.
visibleItemCountLabel
   attachLeft: 4 relativeTo: XmATTACHFORM;
   attachTop: 60 relativeTo: XmATTACHFORM;
   yourself.
visibleItemCount
   attachLeft: 140 relativeTo: XmATTACHFORM;
   attachTop: 60 relativeTo: XmATTACHFORM;
   yourself.
visible
   attachRight: 88 relativeTo: XmATTACHFORM;
   attachTop: 60 relativeTo: XmATTACHFORM;
   yourself.
enabled
   attachRight: 4 relativeTo: XmATTACHFORM;
   attachTop: 60 relativeTo: XmATTACHFORM;
   yourself.
items
   attachLeft: 8 relativeTo: XmATTACHFORM;
   attachRight: 4 relativeTo: XmATTACHFORM;
   attachTop: 92 relativeTo: XmATTACHFORM;
   attachBottom: 42 relativeTo: XmATTACHFORM;
   yourself.
ok
   attachLeft: 8 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   addCallback: XmNactivateCallback
      receiver: self
      selector: #ok:clientData:callData:
      clientData: nil;
   yourself.
```

```
cancel
   attachLeft: 92 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   addCallback: XmNactivateCallback
      receiver: self
      selector: #cancel:clientData:callData:
      clientData: nil;
   yourself.
apply
   attachLeft: 176 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   addCallback: XmNactivateCallback
      receiver: self
      selector: #apply:clientData:callData:
      clientData: nil;
   yourself.
generic
   attachRight: 8 relativeTo: XmATTACHFORM;
   attachBottom: 8 relativeTo: XmATTACHFORM;
   addCallback: XmNactivateCallback
      receiver: self
      selector: #generic:clientData:callData:
      clientData: nil;
   yourself.
```

You should notice that other than code that is inherited from WbAbstractAttributeEditor or directly generated by WindowBuilder Pro, there is no additional work that needs to be done. For some examples of custom editors that do supply additional behavior, examine the other WbAbstractAttributeEditor subclasses supplied with WindowBuilder Pro (e.g., WbLabelEditor, WbScrolledTextEditor, WbPushButtonEditor, etc.).

Using Add-In Modules

Once a widget has been enabled to the WindowBuilder Pro environment via defining its attributes and building a custom editor, it is useful to add it to the tool palette. Add-In modules provide a mechanism for adding widgets to the palette, adding menus to WindowBuilder Pro itself, defining new WindowBuilder Pro properties, and enhancing the default code generation.

The WbAbstractAddInModule class provides the default protocols for all add-ins. Creating a new add-in is a simple matter of subclassing WbAbstractAddInModule and overriding one or more of its protocols.

Four protocols define what aspects of the system this add-in affects:

- **modifiesCodeGeneration** - Does this add-in modify code generation?
- **modifiesMenus** - Does this add-in modify the menus?
- **modifiesPalette** - Does this add-in modify the palette?
- **modifiesProperties** - Does this add-in modify properties?

Just return true for any of these you wish to affect. Four additional protocols are provided that actually do the work:

- **modifyCodeGeneration:** *moduleCollection* - Modify the code generation.
- **modifyMenus:** *aMenuBar* - Modify the menus.
- **modifyPalette:** *thePalette* - Modify the palette.
- **modifyProperties:** *theProperties* - Modify the properties.

Several other protocols are also provided that will be automatically invoked by the system:

- **addInName** - Answer the name of the add-in.
- **addInDescription** - Answer a description of the add-in.
- **addToBitmapModuleMap:** *aDictionary* - Add to the bitmap module map.
- **addToButtonModuleMap:** *aDictionary* - Add to the button module map.
- **cleanUpOnUnload** - Clean up when unloading.
- **initializeOnLoad** - Perform initializations of loading.

The state of an add-in (e.g., whether it is loaded or unloaded) can be set via the `#setLoaded:` or `#loaded:` methods.

As an example, examine the add-in that provides support for VisualAge within WindowBuilder Pro.

```
WbAbstractAddInModule subclass: #WbVisualAgeAddInModule
 instanceVariableNames: ‘’
 classVariableNames: ‘’
 poolDictionaries: ‘’!

!WbVisualAgeAddInModule class publicMethods !

addInName
   “Answer the name of the Add-In”
   ^’VisualAge Support’!
```

```
addInDescription
   “Answer a description of the Add-In”
   ^’Provides support for VisualAge integration’.!

modifiesMenus
   “Does this add-in modify the menus?”
   ^true!

modifyMenus: aMenuBar
   “Add the VisualAge menu to aMenuBar.”
   aMenuBar
      addMenu: (
         aMenuBar owner newMenu
            title: ‘~VisualAge’;
            owner: aMenuBar owner;
            add: #menuVisualAgeAttributeEditor
               label: ‘A~ttribute Editor...’;
            add: #menuVisualAgeActionEditor
               label: ‘~Action Editor...’;
            add: #menuVisualAgeEventEditor
               label: ‘~Event Editor...’;
            yourself).! !

!WbVisualAgeAddInModule class privateMethods !
initialize
   “Private - Initialization sets my loaded state.”
   self loaded: true.
```

This add-in notifies the system that it is interested in modifying the WindowBuilder Pro menubar and then does so via the `#modifyMenus:` method. The menu methods themselves are added as extensions to the WindowBuilder class itself (although they could be anywhere else).

The second example illustrates adding new widgets to the tool palette and the Add menu:

```
WbAbstractAddInModule subclass: #SampleAddInModule
 instanceVariableNames: ‘‘
 classVariableNames: ‘‘
 poolDictionaries: ‘‘!

! SampleAddInModule class publicMethods !

addInName
   “Answer the name of the Add-In”
   ^’Sample Add-In #1’!
```

```
addInDescription
   “Answer a description of the Add-In”
   ^’Description of Sample Add-In #1’.!

addToButtonModuleMap: aDictionary
   “Add to the button module map”
   aDictionary
      at: self module
      put: (IdentityDictionary new
         at: #SampleWidgets put: 100;
         at: #SampleWidget1 put: 101;
         at: #SampleWidget2 put: 102;
         yourself).!

cleanUpOnUnLoad
   “Clean up when unloading”
   self editorClass
      initializeBitmapMaps;
      initializePalette!

initializeOnLoad
   “Perform initializations of loading”
   self editorClass
      initializeBitmapMaps;
      initializePalette!

modifiesPalette
   “Does this add-in modify the palette?”
   ^true!

module
   “Answer the resource DLL”
   ^’SAMPLE.DLL’!

modifyPalette: thePalette
   “Modify the palette”
   thePalette
      add: ((self editorClass paletteGroup
         label: #SampleWidgets mnemonic: $S
         description: ‘‘)
         add: (self editorClass paletteEntry
            label: #SampleWidget1 mnemonic: $1
            description: ‘‘);
         add: (self editorClass paletteEntry
            label: #SampleWidget2 mnemonic: $2
            description: ‘‘);
         yourself);
      yourself! !
```

This add-in loads three additional pixmaps into WindowBuilder Pro’s pixmap cache named “SampleWidgets”, “SampleWidget1” and “SampleWidget2”. The first represents a new widget category. The latter two are the names of the two widgets that are being added to the system. The pixmaps are loaded as bitmap resources from a resource-only DLL that you create (here it is referred to as “SAMPLE.DLL”). Each toolbar bitmap in the DLL must be 21 pixels wide by 21 pixels tall. Use either the Windows or OS/2 RC.EXE utility to create the resource DLL.

The example also notifies the system that it is interested in modifying the palette (which also modifies the Add menu) and does so via the `#modifyPalette:` method. If the `#addToButtonModuleMap:` method was not included above and no pixmaps were added to the cache, a generic widget button would be used instead.

The final example illustrates defining new properties. These properties will be editable from within the Property Editor. The current values of the properties can be used in conjunction with other add-ins that modify the WindowBuilder Pro menus or code generation.

```
WbAbstractAddInModule subclass: #SampleAddInModule2
 instanceVariableNames: ‘‘
 classVariableNames: ‘‘
 poolDictionaries: ‘‘!

! SampleAddInModule2 class publicMethods !
addInName
   “Answer the name of the Add-In”
   ^’Sample Add-In #2’!

addInDescription
   “Answer a description of the Add-In”
   ^’Description of Sample Add-In #2’!

cleanUpOnUnLoad
   “Clean up when unloading”
   self editorClass initializeProperties!

initializeOnLoad
   “Perform initializations of loading”
   self editorClass initializeProperties!

modifiesProperties
   “Does this add-in modify the properties?”
   ^true!
```

```
modifyProperties: theProperties
   “Modify the properties”
   theProperties
      at: self samplePropertyString
      put: (IdentityDictionary new
         at: #BooleanProperty
         put: (WbPropertyDescriptor new
            name: #BooleanProperty;
            commonName: ‘Boolean Property’;
            comment: ‘Description of property.’;
            category: self samplePropertyString;
            default: true;
            changeBlock: [:newValue | “do something”];
            yourself);
         at: #StringProperty
         put: (WbPropertyDescriptor new
            name: #StringProperty;
            commonName: ‘String Property’;
            comment: ‘Description of property.’;
            category: self samplePropertyString;
            default: ‘default string’;
            yourself);
         at: #PointProperty
         put: (WbPropertyDescriptor new
            name: #PointProperty;
            commonName: ‘Point Property’;
            comment: ‘Description of property.’;
            category: self samplePropertyString;
            default: 0@0;
            yourself);
         yourself);
      yourself.!

samplePropertyString
   ^#’Sample Properties’
```

This add-in notifies the system that it is interested in modifying the WindowBuilder Pro property list and then does so via the `#modifyProperties:` method. Three add-ins of different types are defined in a new category. An optional changeBlock may be specified that takes the new value of the property as an argument.

Note that any single add-in module can actually affect multiple aspects of the system. The three examples above could be easily combined into a single add-in that modifies the WindowBuilder menu bar, property list and tool palette.

Adding Code Generation

The WindowBuilder Pro code generation framework is also highly extensible. The framework is composed of the WbCodeGenerator class that acts as a general coordinator for multiple subclasses of WbCodeModule. There exists one subclass of WbCodeModule for each type of method that WindowBuilder Pro can generate. Extending the code generation framework involves adding new WbCodeModule subclasses and then tying them into the WbCodeGenerator class.

When subclassing WbCodeModule, the following protocols are of interest:

category **[class method]**
Answer the primary category in which the method should be included. The default is “WBPro-Generated”.

defaultStreamSize
Answer the default stream size to be used. The default is 256 bytes. This is used to set the initial size of the stream.

generateBody
Generate the body of the method. This method executes after the message pattern and any temporaries are defined. This is where most of the work is done. For stub methods such as callback handlers, this method should do nothing. Look at the WbWidgetDefinitionCodeModule and WbMenuDefinitionCodeModule classes for examples.

generateCommentBody
Generate the comment body for the method. By default, this will generate the comment “Generated by WindowBuilder Pro”.

generateCommentBodyIndent: *indent*
Generate the comment body for the method indented by *indent*. Having this method invoke `#generateWarningCommentBodyIndent:` will generate the standard WindowBuilder Pro warning message.

generateTemporaries
Generate the temporaries for the method. Look at the WbWidgetDefinitionCodeModule class for an example.

initializeMethod
Perform special initializations before the method is generated.

isMeta
Should the method be a class method? The default is false. Override this method and return true for class methods.

isPublic
Should the method be public? The default is false. Override this method and return true to create a public method.

methodArguments
Answer the collection of method arguments. For each key word in the generated method, there should be one argument provided. Look at the WbCallbackCodeModule class for an example.

methodName
Answer the method name (as a Symbol) to be generated. All of the WbCodeModule classes override this method.

postInitializeBody
Perform special initializations after the body of the method is generated.

preInitializeBody
Perform special initializations before the body of the method is generated.

In addition to these protocols, there are several helper methods that you can use:

codeGenerator
Answer the value of the codeGenerator instance variable. This is the instance of WbCodeGenerator that is managing the current code module.

object
Answer the object for which code is being generated. This object is held by the WbCodeGenerator instance that is managing the current code module.

policy
Answer the code storage policy instance. The code storage policy instance is responsible for taking the generated method stream and storing it in the base system. The code storage policy is maintained by the WbCodeGenerator instance that is managing the current code module.

shouldStore
Answer the value of the shouldStore instance variable. This is a flag that indicates whether the method should actually be saved or not. It is possible that during the course of generating a method, an error is discovered or the method is empty and does not need to be saved. Setting this flag to false will prevent the method from being saved.

storageClass
Answer the class in which to create the method. This should be the class of the object for which code is being generated.

stream
Answer the value of the stream instance variable. This is the current stream that the method is being generated to.

After creating your WbCodeModule subclass, you need to add a class method to WbCodeGenerator that references it. It should take the following form:

```
generateSomeCodeFor: anObject
   “Generate code for <anObject>”
   self
      generateCodeFor: anObject
      using: SomeCodeModule.
```

Examine the class methods of WbCodeGenerator for several examples. You can now generate code via:

```
WbCodeGenerator generateSomeCodeFor: anObject
```

Appendix B Extended Widgets

Common Widgets provides a framework for developing custom widgets based on existing widgets. These are called extended widgets. If the VisualAge portable API is used to develop an extended widget, it will be portable between all platforms supported by VisualAge. Extended widgets are often implemented using a CwDrawingArea, with their visual appearance drawn using Common Graphics calls, and with user input processed using event handlers.

Consider the following subset of the CwWidget class hierarchy.

- **CwWidget**
  - **CwBasicWidget**
    - **CwComposite**
    - **CwPrimitive**
    - **CwShell**
  - **CwExtendedWidget**
    - **CwExtendedComposite**
    - **CwExtendedPrimitive**

The CwWidget class defines behavior common to all widgets. The CwBasicWidget hierarchy provides the basic widgets described thus far, such as CwShell, CwText, CwList, CwPushButton, CwForm and CwRowColumn. Basic widgets are implemented using the native widgets provided by each platform. The implementation of basic widgets is not portable.

The CwExtendedWidget class is the abstract superclass of all extended widgets. As with the basic widget class hierarchy, it is divided up into primitive widgets (CwExtendedPrimitive) and composite widgets (CwExtendedComposite).

Writing an Extended Widget

The first step in writing an extended widget is to create a subclass of the appropriate extended widget framework class. Extended widgets that are not intended to contain child widgets should be implemented as subclasses of CwExtendedPrimitive. Those that are intended to contain child widgets should be implemented as subclasses of CwExtendedComposite. It is important to understand this difference.

A subclass of CwExtendedPrimitive can be implemented using a primary widget with child widgets, however an application programmer making use of this type of extended widget cannot add any children to it.

A subclass of CwExtendedComposite can be implemented using just a single widget, say for example a CwForm with no children, but the same application programmer can create this type of extended widget and add as many children as are permitted by the extended widget's API.

Once the subclass has been created, it should define an instance variable for each resource and callback provided by the extended widget, as well as instance variables required for any other aspects of the widget's implementation.

Defining the Extended Widget Class

An extended widget is implemented using a widget tree consisting of other basic or extended widgets. This tree is called the primary widget tree. The root of the primary widget tree is known as the primary widget. The extended widget class must override the `#createPrimaryWidget:parent:argBlock:` method. This method creates and answers the primary widget, but does not create the children of the primary widget. If the primary widget tree consists of more than one widget, the extended widget class must override `#createWidgetSystem`. This method creates the remainder of the primary widget tree, that is, the children of `self primaryWidget`.

Initialization

Three methods can be overridden to initialize the state of the widget. The initialize method is executed as the first step in extended widget creation. It is useful for initializing the internal state of the widget, except for resources. The `#initializeResources` method initializes the instance variables representing resources. Both of these methods are executed before the primary widget tree has been created. The `#initializeAfterCreate` method is executed after the primary widget tree has been created. It is useful for configuring widgets once they have been created, and for initializing graphics resources.

Resources

Set and get accessor methods should be added for each resource provided by the extended widget. Usually, the get method simply answers the corresponding instance variable. The set method usually sets the corresponding instance variable and makes any required changes in the primary widget tree.

Callbacks

Set and get accessor methods must be added for each callback provided by the extended widget. To work properly with methods inherited from CwExtendedWidget, the get method name must have the same selector as the value of the CwConstants constant used to specify the callback, for example, exposeCallback for the XmNexposeCallback. If a new name is created, it can be added to an application-specific pool dictionary, but the same naming convention must be used. It must answer an ordered collection, to which callback descriptors are added whenever a callback is registered. If the callback resource is uninitialized, the get method must initialize the callback resource to be a new OrderedCollection, and answer that. Registered callbacks can be executed by the extended widget using the `#callCallbacks:callData:` method.

Widget-Specific Methods

An extended widget works by forwarding most widget messages to its primary widget. All of the methods inherited from CwWidget are automatically forwarded to the primary widget if they are not explicitly overridden. In simple cases, an extended widget's behavior can be implemented simply by adding resource and callback methods as described above. For more complicated widgets, it is usually necessary to extend the basic widget protocol by providing methods to support commonly used operations on the extended widget.

Using an Extended Widget

Once an extended widget class has been defined, application developers can create instances of the extended widget by sending the `#createWidget:parent:argBlock:` or `#createManagedWidget:parent:argBlock:` method to the extended widget's class. The create argBlock should only set resources that are defined for the extended widget or in CwWidget, and should not assume any underlying implementation.

Example: A Primitive Extended Widget

An example extended widget is provided in this section. It is a subclass of CwExtendedPrimitive. For simplicity, the example does not include robust error-checking, nor does it provide a complete set of resources.

The WbLabelledText widget has a label on either the left or top and a text box either on the right or bottom. It allows a user to enter text into its text box, and it invokes a #valueChanged callback if a new value is present when the user either hits the tab key or clicks on a different widget. The complete code for this example is in the WbProRuntimeExamples application.

The extended widget is implemented using a CwForm as the primary widget with CwLabel and CwText children. A #losingFocus callback on the CwText enables the widget to test entered data, and possibly call any registered #valueChanged callbacks.

```
CwExtendedPrimitive subclass: #WbLabelledText
    instanceVariableNames: ‘label value format
       valueChangedCallback textWidget labelWidget ‘
    classVariableNames: ‘‘
    poolDictionaries: ‘‘!

backgroundColor: resourceValue
   “Specifies the background drawing color.”
   super backgroundColor: resourceValue.
   self children do: [:child |
      child backgroundColor: resourceValue].

children
   “Private - answer the receivers children”
   ^self primaryWidget children

createPrimaryWidget: theName parent: parent argBlock: argBlock
   “Private - Create and answer the basic widget that is the
    root of the widget hierarchy for the receiver’s widget
    system.”
   ^self parent
      createForm: theName, ‘_Form’
      argBlock: argBlock

createWidgetSystem
   “Private - Create the children of the receiver’s primary
    widget which form the widget hierarchy.”
   | primaryWidget |
   primaryWidget := self primaryWidget.
   labelWidget := primaryWidget
      createLabel: primaryWidget name, ‘Label’
      argBlock: [:w | w
         labelString: self label].
   labelWidget manageChild.
   textWidget := primaryWidget
      createText: primaryWidget name, ‘Text’
      argBlock: [:w | w
```

```
            borderWidth: 1;
            value: self value].
    textWidget
        addCallback: XmNlosingFocusCallback
            receiver: self
            selector: #losingFocus:clientData:callData:
            clientData: nil;
        manageChild.
    labelWidget setValuesBlock: [:w | w
        topAttachment: XmATTACHFORM;
        leftAttachment: XmATTACHFORM].

foregroundColor: foreground
    “Specifies the background drawing color.”
    super foregroundColor: resourceValue.
    self children do: [:child |
        child foregroundColor: resourceValue].

format
    “Answer the layout format.”
    ^format

format: aSymbol
    “Set the value of the format to aSymbol.”
    (#(column row) includes: aSymbol)
        ifTrue: [
            format := aSymbol.
            self isDestroyed
                ifFalse: [
                    self
                        setTextWidgetAttachment;
                        setLabelWidgetFormat]].

initializeAfterCreate
    “Private - Perform widget specific Initialization.”
    self
        setTextWidgetAttachment;
        setLabelWidgetFormat.

initializeResources
    “Private - Set the default extended widget resource values.
     This is sent during create with isCreated set to false.
     All extended resource variables should be initialized
     to default values here.”
    label := String new.
    value := String new.
    format := #row.
```

```
label
   “Answer the value of the label resource.”
   ^label

label: aString
   “Set the value of the label resource to aString.”
   label := aString.
   self isDestroyed
      ifFalse: [self setLabelWidgetFormat]

losingFocus: widget clientData: clientData callData: callData
   “Private - Catch losing focus callback and
    call value changed callback”
   | textValue |
   (textValue := textWidget value) ~= self value
      ifTrue: [
         self
            value: textValue;
            callCallbacks: XmNvalueChangedCallback
            callData:
               (CwValueCallbackData new value: textValue)].

setLabelWidgetFormat
   “Private - Format labelWidget corresponding to format”
   format == #column
      ifTrue: [labelWidget labelString: self label]
      ifFalse: [
         labelWidget
            labelString: self label, ‘: ‘;
            height: textWidget height].

setTextWidgetAttachment
   “Private - Position textWidget corresponding to format”
   format = #column
      ifTrue: [
         textWidget setValuesBlock: [:w | w
            topAttachment: XmATTACHWIDGET;
            topWidget: labelWidget;
            leftAttachment: XmATTACHFORM;
            rightAttachment: XmATTACHFORM]]
      ifFalse: [
         textWidget setValuesBlock: [:w | w
            topAttachment: XmATTACHFORM;
            leftAttachment: XmATTACHWIDGET;
            leftWidget: labelWidget;
            rightAttachment: XmATTACHFORM]].
```

```
value
   “Answer the value of the value resource.”
   ^value

value: aString
   “Set the value of the value resource to aString.”
   value := aString.
   self isDestroyed
      ifFalse: [textWidget value: aString]!

valueChangedCallback: resourceValue
   “Set valueChangedCallback to resourceValue.”
   valueChangedCallback := resourceValue.

valueChangedCallback
   “Private - Answer valueChangedCallback.”
   valueChangedCallback
      ifNil: [self valueChangedCallback: OrderedCollection new].
   ^ valueChangedCallback
```

Using the WbLabelledText Primitive Extended Widget

The following code creates a WbLabelledText instance, sets its name and label resources inside the create argBlock, and hooks a valueChanged callback to it.

```
   | shell entryField |
   shell := CwTopLevelShell
      createApplicationShell: ‘WbLabelledText Test’
      argBlock: nil.
   entryField := WbLabelledText
      createManagedWidget: ‘entryField’
      parent: shell
      argBlock: [:w |
         w
            label: ‘Name :’;
            value: ‘Your name here’].
   entryField
      addCallback: XmNvalueChangedCallback
      receiver: self
      selector: #valueChanged:clientData:callData:
      clientData: nil.
   shell realizeWidget.

valueChanged: widget clientData: clientData callData: callData
   “Display the new value on the transcript.”
   Transcript cr; show: ‘Value changed to: ‘ ,
      callData value printString.
```

The WbLabelledText class can be subclassed to provide a slightly different extended widget by simply overriding one method, as seen below in the class definition for WbLabelledNumericText.

```
WbLabelledText subclass: #WbLabelledNumericText
 instanceVariableNames: ‘‘
 classVariableNames: ‘‘
 poolDictionaries: ‘‘

losingFocus: widget clientData: clientData callData: callData
   “Private - Process a losing focus callback for the
    primary widget.”
   | textValue |
   textValue:= textWidget value.
   “Verify that the new value string represents a number.
    If it doesn’t, reset the text widget and return.”
   (newValue notEmpty and: [newValue conform: [:c | c isDigit]])
      ifFalse: [^self value: value].
   “If the new value is different, invoke the
    entryField widget’s valueChanged callback.”
   self value ~= textValue
      ifTrue: [
         self
            value: textValue;
            callCallbacks: XmNvalueChangedCallback
            callData:
                   (CwValueCallbackData new value: textValue)]
```

Appendix C User Interface Process Model

The Common Widgets user interface has been modeled based on the input event processing model supported by OSF/Motif. A central event processing loop reads events from the operating system and dispatches them to individual widgets that process them. User interfaces (GUIs) including OSF/Motif, Microsoft Windows, IBM's OS/2 Presentation Manager, and the Apple Macintosh operating system use this technique.

- In Common Widgets, the event polling loop has been implemented fully within high-level Smalltalk code. This has a number of significant benefits to the application programmer:
- Existing Motif application programmer knowledge is maintained, since custom event loops can be constructed in the standard Motif style without fear of error-causing interactions with hidden event handling mechanisms.
- Event polling occurs only at controlled points during system execution, so application code runs at maximum speed.
- The system requires none of the cumbersome and error-prone low-level synchronization code required by systems that attempt to hide the event loop below the control of the application programmer.
- Complex multi-threaded applications can safely perform work in background processes, while the user interface process continues to keep the user interface responsive.

With Common Widgets, only the single Smalltalk user interface process is permitted to dispatch events or directly perform user interface operations. Common Widgets facilitates a proactive approach to event management, as opposed to a defensive one. Rather than write code to defend themselves against asynchronous user interface events, such as exposes, menu operations, or user input, application programmers control the user interface event processing. Event processing is fully synchronous from the perspective of the user interface process, although it can be asynchronous from the perspective of non-UI processes.

Unfortunately, as is often the case, increased capability comes at the cost of increased responsibility. In the case of the polled event model, application programmers are responsible for writing their applications in ways that allow polling to occur at frequent intervals. The responsiveness of an application—that is, the delay between the availability of an event and processing of the event by the application—is directly affected by the frequency at which the application polls. Although they vary in their sensitivity to failures, all of the GUIs mentioned above specify that frequent polling is required to maintain application responsiveness.

Common Widgets provides support for maintaining application responsiveness while long-running tasks execute. This support is based on the Common Process Model together with a standard application program interface (API) for managing the interactions between non-UI tasks and the user interface. This is discussed in detail in the following sections. First, a system view is presented, which provides an overview of the implementation of these mechanisms, and then an application view is presented, which discusses how the mechanisms are used in building applications.

The System View

In Smalltalk images that include a user interface (Common Widgets), the startUp class (System startUpClass) is responsible for providing a polling loop for the user interface process, an instance of UIProcess. The UIProcess sends the message `#messageLoop` to the startUp class to start the polling loop. The standard startUp class, EtWindowSystemStartUp, implements a simple polling loop similar to the one shown below:

```
messageLoop
   “Run the dispatch loop.”
   [true] whileTrue: [
      CwAppContext default readAndDispatch
         ifFalse: [ CwAppContext default sleep ]
   ]
```

In general, application programmers never need to modify this code because it provides full functionality for all but the most exceptional circumstances. However, as mentioned above, application programmers can replace this loop with their own.

The message loop makes use of two methods defined in class CwAppContext:

readAndDispatch Reads a single event, if one is available, from the underlying operating system, dispatches it to the appropriate widget, and handles any callbacks that occur. In addition, it handles any pending requests for user interface operations by non-UI processes, as shown below. Finally, it returns true if an event was processed and false otherwise.

sleep Checks for user interface activity, and if none, removes the UIProcess from the ready-to-run queue. The system assumes there is user interface activity in the following cases:

- There are events to process
- There are background user interface requests to be executed
- There are work procs registered
- There are timer procs registered

As long as there is any activity in the user interface, the UIProcess will continue to poll for events as quickly as possible. As soon as the activity stops, the UIProcess becomes inactive and suspends. This enables any other Smalltalk processes that are running at the same or lower priority than the UIProcess to execute.

Because sending the CwAppContext default sleep method can deactivate the UIProcess, there must be a mechanism for reactivating it. To support this, sleep enables an operating system-specific mechanism that causes the private message CwAppContext default wake to be sent when new events become available. This wake method is also sent by all other methods that generate user interface activity, causing the UIProcess to respond immediately.

If the underlying operating system does not provide any mechanism for triggering user written code when events become available, the CwAppContext can still function by generating a Smalltalk process that wakes the UIProcess at regular, user-settable intervals. By default, this is called the “CwAsyncIOProcess”.

As previously mentioned, if there is no user interface activity the UIProcess is deactivated, enabling other Smalltalk processes at the same or lower priority to run. However, if there are no other active processes to run, a system-provided “idle” process is executed, which repeatedly sends the `#suspendSmalltalk` message to the default CwAppContext:

suspendSmalltalk Suspends the entire VisualAge system, using an operating system-specific facility, until there is event activity. When the VisualAge system is suspended it consumes little or no processor resources. Under multi-tasking operating systems this enables other applications full access to the CPU.

As soon as input is available, both the VisualAge system and the UIProcess are reactivated, because they are higher priority than the idle process,.

If the operating system does not provide a facility for suspending execution of an application until input is available, the `#suspendSmalltalk` method simply returns to its sender. In this case, Common Widgets continues to run normally, but VisualAge competes for resources with any other applications being run by the operating system.

Note that there is an interaction between the `#suspendSmalltalk` method and the Smalltalk Delay class. If the idle process runs because a higher priority process has delayed, the system must be reactivated when the Delay expires. This situation is handled in one of three ways depending on the capabilities of the operating system:

- With operating systems such as UNIX where the Delay class uses the same mechanism that `#suspendSmalltalk` uses to detect input activity, the system is reactivated with no further intervention.
- With operating systems where Delay uses a different mechanism than `#suspendSmalltalk`, but it is possible for a user written application to post a user interface event, this facility is used to reactivate the system.
- If neither of the above mechanisms are available, VisualAge checks for Delays and deactivates the system only when there are none pending.

The Application Programmer’s View

This section looks at the impact of input polling on application code.

When programming with OSF/Motif, or other modern window systems, application programmers writing simple applications typically do not worry much about the details of the input model. They simply define the kinds and layout of the widgets that make up their application windows, register the appropriate callbacks, or equivalent, for user actions, and implement the code for these callbacks. Although this is ideally true, one important aspect is sometimes not considered: because callbacks are executed by the same process that is running the user interface, all of the code that is executed by the callback—an operation—is executed before the application returns to polling. Thus, when long-running operations are executed, application responsiveness, and whole system responsiveness on some operating systems, is affected.

In Common Widgets, programmers construct applications using the standard OSF/Motif model described above. However, additional facilities are provided to allow long-running operations to execute without impacting responsiveness.

Because VisualAge provides support for multiple, priority-scheduled threads of control, or processes, operations that do not use user interface facilities can be executed by a separate, low-priority process, called a background process. The user interface remains responsive because it is higher priority than the background process and is activated whenever input is available.

Unfortunately, OSF/Motif itself does not use a multi-threaded model and thus it is not possible for multiple application processes to concurrently execute user interface code, for example, redrawing a widget, performing an individual graphical operation, or reading an event. Some kind of synchronization is necessary.

It is possible for each application to implement its own synchronization scheme that prevents background processes from attempting to concurrently execute user interface code. However, VisualAge provides standard mechanisms to support this kind of synchronization.

A background user interface request is a block of code that must be executed by the UIProcess at a point when no other user interface code is being executed. The following two methods are then implemented in class CwAppContext to take background user interface requests.

Note: A background (non-UI) process must never make widget or graphics requests directly—it must use `#syncExecInUI:` or `#asyncExecInUI:` to ask the user interface to issue the request on its behalf.

Background user interface requests are executed by atomically interleaving their execution with the event and callback processing of the user interface process. Once execution of the block has been started, no further user interface events can be processed until execution of the block has been completed.

asyncExecInUI: The UIProcess executes aBlock during `#readAndDispatch` processing, at the next "clean" point, that is, at the next point where it is not already executing user interface code. No result is returned.

Processes with higher priority than the UIProcess do not block when this method is executed. In this case, aBlock is executed the next time the UI becomes active and reaches a clean point. Processes at the same or lower priority than the UIProcess normally block until aBlock has been executed, but this is not guaranteed.

The process that executes this message is not provided any indication of when aBlock has been executed. In particular, it is not guaranteed that aBlock has been executed when the method returns.

If this message is sent by code that is executing in the UIProcess, then aBlock is executed after all previously queued background user interface requests have been executed.

Tip: A background process can re-awaken the UIProcess and cause a context switch (since the UIProcess is higher priority) by executing: `CwAppContext default asyncExecInUI: []`

syncExecInUI: aBlock is executed by the UIProcess during `#readAndDispatch` processing, at the next “clean” point, that is, at the next point where it is not already executing user interface code. The result of evaluating aBlock is returned.

Execution of the process that evaluates this method always suspends until aBlock has been evaluated.

If this message is sent by code that is executing in the UIProcess, the block is executed immediately.

Both of these methods are implemented to add the code to be executed, aBlock, to a collection of background user interface requests maintained by the CwAppContext. This collection is FIFO ordered so operations can use `#syncExecInUI:` to ensure that all previously sent `#asyncExecInUI:` messages have been processed.

After adding the request, both of the above methods send the message CwAppContext default wake to re-activate the UIProcess if it was sleeping. As was described in the previous section, the CwAppContext processes the background user interface requests as part of the normal functioning of the readAndDispatch method. Processing of pending background user interface requests is interleaved with user interface event dispatching.

Using the above methods, it is possible for programmers to construct applications that are both responsive and contain long-running operations. The long-running operations are executed by background tasks that use `#asyncExecInUI:` or `#syncExecInUI:` to access the user interface. Obviously, the programmer is responsible for ensuring that the individual blocks that are passed to the “ExecInUI” methods do not take an unusually long time to execute.

Note: Since execution of background graphics requests can be deferred while events are processed, background user interface request blocks must be prepared for eventualities such as widgets being destroyed between the request being posted and executed.

Examples of Applications with Long-Running Operations

The following are some overviews of correct implementation styles when building applications with long-running operations using VisualAge. These examples are intended to show typical ways that applications can be written using VisualAge. Of course, every application is different so developers can use any or all of these techniques, or construct new idioms that are more appropriate to their problem domain. The important point to remember is that there are no surprises. Every aspect of the polling model is accessible to the application developer. Nothing is done “under the covers.”

Example 1: A Simple Text Editor

A simple text editing application is constructed using VisualAge. The application is responsive in all situations except while reading or writing the file that is being edited. To maintain responsiveness in this situation, file reading and writing are moved to a background process. The operations are modified to use a modal “percentage complete” dialog that is updated by the background process using `#asyncExecInUI:`. When the file has been completely read or written, the background process uses a call to `#syncExecInUI:` to close the dialog.

Example 2: A Program Development Environment

A program development environment is constructed that uses a database to store source code. Saving large segments of source code is found to cause a lack of responsiveness because several database accesses are required. To maintain responsiveness, the saving operation is modified to:

- Disable any menus or buttons that could affect the saving operation
- Change the cursor to indicate that an operation is “in progress”
- Generate a background process that first saves the source to the database, and then uses `#asyncExecInUI:` to re-enable the menus and buttons and set the cursor back to normal.

Example 3: A Complex Drawing Editor

A drawing editor is constructed that allows large, complex drawings to be built. The system is responsive in all situations except while updating a display of a large drawing. The update display (refresh) operation is modified as follows:

- First, it checks the complexity of the drawing (for example, by detecting the total number of primitive graphical objects that it contains) and if it is below a particular threshold, it refreshes the drawing directly.
- If the drawing is too complex to refresh in the UIProcess, the operation changes the cursor to indicate that a refresh operation is “in progress.”

- Then it generates a background process that draws the diagram, one component at a time, using a background user interface request for each component. After the diagram is redrawn, the application sets the cursor back to its standard shape using a synchronous background user interface request. The application retains a reference to the background process in order to terminate it if the user performs some action that invalidates its usefulness (for example, resizing or closing the window, or using a menu entry to request another refresh or switch to a different drawing).

Notice that in this example, the same code could be used to do the actual refreshing of the drawing, regardless of whether it is being executed by the UIProcess or by a background process, since the “ExecInUI” methods can always be called by both background processes and the UIProcess.

Appendix D Common Widgets Platform Differences

Parts of the Common Widgets subsystem can behave differently depending on constraints of the platform (hardware, operating system and window system). For example, Windows supports only one alignment for text in button widgets. Where possible, Common Widgets features are mapped to the closest available on the platform.

Windows and OS/2 Platform Differences

The tables below identify the platform constraints of the Common Widgets subsystem under Windows and OS/2. Blank cells indicate that the corresponding item is fully supported for the indicated platform.

General		
Item	**DOS/Windows**	**OS/2 PM**
Border width	The borderWidth resource can only be 0 or 1 pixels	The borderWidth resource can only be 0 or 1 pixels
Background and foreground color	Only solid colors are supported. Colors are not dithered.	

Arrow Button Widgets (CwArrowButton)		
Item	**DOS/Windows**	**OS/2 PM**
Sizing	Displays the message ‘Arrow too big’ when the widget is grown such that the bitmap that is used to draw the arrow becomes larger than 64K	
Border width		The borderWidth resource is ignored

Button and Label Widgets

(CwLabel, CwPushButton, CwToggleButton, CwCascadeButton, CwDrawnButton)

Item	DOS/Windows	OS/2 PM
Alignment	The alignment resource is ignored for CwPushButton and CwToggleButton. The label is placed by the OS.	The alignment resource is ignored for CwPushButton and CwToggleButton. The label is placed by the OS.
Margins	The following resources affect only the total width and height of the widget, not the positioning of the label or pixmap inside the widget: marginBottom,marginHeight, marginLeft, marginRight, marginTop, marginWidth	The following resources affect only the total width and height of the widget, not the positioning of the label or pixmap inside the widget: marginBottom,marginHeight, marginLeft, marginRight, marginTop, marginWidth
Default shadow width	The showAsDefault resource can only be 0 or 1	The showAsDefault resource can only be 0 or 1
Border width		The borderWidth resource is ignored
As menu items	The following are not supported: backgroundColor, foregroundColor, tab traversal and focus control (setInputFocus, navigationType, traverseOn), and geometry requests (and the geometry values are undefined). The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget	The following are not supported: backgroundColor, foregroundColor, tab traversal and focus control (setInputFocus, navigationType, traverseOn), and geometry requests (and the geometry values are undefined). The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget
Color	Cannot set backgroundColor or foregroundColor	

Combo Box Widgets

(CwComboBox)

Item	DOS/Windows	OS/2 PM
Event handlers	Event handlers can be hooked on the combo box widget but not on the children that implement the combo box	Event handlers can be hooked on the combo box widget but not on the children that implement the combo box
Editing	When the comboBoxType resource is XmSIMPLE, the combo box is always editable. The editable resource has no effect	When the comboBoxType resource is XmSIMPLE, the combo box is always editable. The editable resource has no effect
Default selection		When the comboBoxType resource is XmDROPDOWN and there is no initial string setting, the first item in the combo box is selected when the combo box is dropped

List Widgets

(CwList & WbScrolledList)

Item	DOS/Windows	OS/2 PM
Selection policy	XmSINGLESELECT behaves the same as XmBROWSESELECT	XmSINGLESELECT behaves the same as XmBROWSESELECT
Size limit	The total number of characters in all list items, plus one for each list item, must be less than 64K	The total number of characters in all list items, plus one for each list item, must be less than 64K
Automatic scrolling	Positioning the list (setting the top and bottom item) may not actually scroll the list. This is because only one blank line is allowed at the bottom of the list and therefore, depending on the number of items, the height of the list and the desired position, the list may not scroll.	Positioning the list (setting the top and bottom item) may not actually scroll the list. This is because only one blank line is allowed at the bottom of the list and therefore, depending on the number of items, the height of the list and the desired position, the list may not scroll.
Scroll bars	When created normally (not as a scrolled list), no scroll bars appear. When created as a scrolled list, the vertical scroll bar is visible only when required. The horizontal scroll bar is visible when the scrollHorizontal resource is true and the list contains an item that is wider than the width of the list.	When created normally (not as a scrolled list), a vertical scroll bar appears. When created as a scrolled list, the vertical scroll bar is always visible. The horizontal scroll bar is visible when the scrollHorizontal resource is true.
Border width		The widget is always displayed with a 1-pixel border.

Main Window Widgets **(CwMainWindow)**		
Item	**DOS/Windows**	**OS/2 PM**
Organization	Must be created as the child of a shell	Must be created as the child of a shell
Scroll bar limitations	Scroll bar children have the following limitations: the help callback is ignored, tab traversal and focus control (setInputFocus, navigationType, traverseOn) are not supported, geometry requests are ignored and the initial geometry values are undefined. The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget	Scroll bar children have the following limitations: the help callback is ignored, tab traversal and focus control (setInputFocus, navigationType, traverseOn) are not supported, geometry requests are ignored and the initial geometry values are undefined. The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget

Menus and Menu Bars

(CwRowColumn with rowColumnType of XmMENUBAR, XmMENUPULLDOWN or XmMENUPOPUP)

Item	DOS/Windows	OS/2 PM
Types of child widgets	Only CwLabel, CwToggleButton, CwSeparator or CwCascadeButton can be added	Only CwLabel, CwToggleButton, CwSeparator or CwCascadeButton can be added
Help callback	Not supported	Not supported
Background color	Not supported	Not supported
Tab traversal and focus control (setInputFocus:, interceptEvents:, grabPointer:, ungrabPointer:, navigationType, traverseOn)	Not supported	Not supported
Geometry requests	Geometry requests are ignored and the initial geometry values are undefined	Geometry requests are ignored and the initial geometry values are undefined
Stacking order	Stacking order requests (bringToFront) do nothing	Stacking order requests (bringToFront) do nothing
Event handlers	Can be hooked but do nothing	Can be hooked but do nothing
Updating widgets	The updateWidget method does nothing	The updateWidget method does nothing
Ignored resources	The following resources are ignored: adjustLast, borderWidth, entryAlignment, entryBorder, marginHeight, marginWidth, numColumns, orientation, packing, spacing	The following resources are ignored: adjustLast, borderWidth, entryAlignment, entryBorder, marginHeight, marginWidth, numColumns, orientation, packing, spacing
Accelerator text with pixmap menu items	When a child has a labelType of XmPIXMAP, its acceleratorText is not shown	When a child has a labelType of XmPIXMAP, its acceleratorText is not shown
Drawn buttons	CwDrawnButton children are not supported	CwDrawnButton children are not supported
Separators	The separatorType, orientation, and margin resources are ignored for CwSeparator children.	The separatorType, orientation, and margin resources are ignored for CwSeparator children.

Menu Bars

(CwRowColumn with rowColumnType of XmMENUBAR)

Item	DOS/Windows	OS/2 PM
Unmapping	Unmapping the widget has the same visual effect as unmanaging	Unmapping the widget has the same visual effect as unmanaging
Parent widget	Can only be created as a child of a CwMainWindow	Can only be created as a child of a CwMainWindow

Menus

(CwRowColumn with rowColumnType of XmMENUPULLDOWN or XmMENUPOPUP)

Item	DOS/Windows	OS/2 PM
Redrawing	The redraw and redraw:y:width:height: methods do nothing.	The redraw and redraw:y:width:height: methods do nothing.

Scale Widgets

(CwScale)

Item	DOS/Windows	OS/2 PM
Event handlers	Event handlers can be hooked on the scale widget but not on the children that implement the scale	Event handlers can be hooked on the scale widget but not on the children that implement the scale

Scrolled Bar Widgets

(CwScrollBar)

Item	DOS/Windows	OS/2 PM
Border width	Ignored	Ignored

Scrolled Window Widgets

(CwScrolledWindow)

Item	DOS/Windows	OS/2 PM
Organization	The work area must be a child of the scrolled window	The work area must be a child of the scrolled window
Scroll bar limitations	Scroll bar children have the following limitations: the help callback is ignored, tab traversal and focus control (setInputFocus, navigationType, traverseOn) are not supported, geometry requests are ignored and the initial geometry values are undefined. The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget	Scroll bar children have the following limitations: the help callback is ignored, tab traversal and focus control (setInputFocus, navigationType, traverseOn) are not supported, geometry requests are ignored and the initial geometry values are undefined. The following methods do nothing: stacking order requests (bringToFront), event handlers (even though they can be hooked), deferRedraw:, and updateWidget

Text Widgets

(CwText & WbScrolledText)

Item	DOS/Windows	OS/2 PM
Highlight vs. selection	Highlight and selection are the same. It is not possible to set the highlight without affecting the selection.	Highlight and selection are the same. It is not possible to set the highlight without affecting the selection.
Highlight appearance	XmHIGHLIGHTSELECTED appears as inverse text	XmHIGHLIGHTSELECTED appears as inverse text
Insertion vs. selection	Insertion and selection are the same. It is not possible to move the insertion point without affecting the selection.	Insertion and selection are the same. It is not possible to move the insertion point without affecting the selection.
Insertion point	The insertion point (cursorPosition, getInsertionPosition) is always answered as the beginning of the selection	
Visual appearance		The selection is hidden when the widget loses focus
Background and foreground color		Only solid background colors are supported

Top Level Shells and Dialog Shells **(CwTopLevelShell, CwDialogShell)**		
Item	**DOS/Windows**	**OS/2 PM**
Window titles	Centered	Left
Behavior of mwmDecorations resource	1) A title bar is always included (MWMDECORTITLE). 2) If MWMDECORMAXIMIZE or MWMDECORMINIMIZE is set, MWMDECORRESIZEH is added. 3) If a menu bar is added to a shell that has MWMDECOR-BORDER, Windows does not paint the shell properly.	No title bar is included unless MWMDECORTITLE is set.

Appendix E VisualAge Integration

This extension adds a VisualAge menu to WindowBuilder Pro and enables WindowBuilder Pro created windows (WbApplication subclasses) to function as VisualAge parts.

Prerequisites

The WindowBuilder Pro/VisualAge integration requires both WindowBuilder Pro and the VisualAge visual tools (e.g., the Composition Editor) as prerequisites. The WindowBuilder Pro/VisualAge integration classes will be automatically loaded when WindowBuilder Pro is installed into an image containing the VisualAge visual tools.

Using WindowBuilder Pro Created Parts In VisualAge

WindowBuilder Pro created windows (WbApplication subclasses) may be used as parts within the VisualAge Composition Editor by adding the windows to a VisualAge canvas as wrappered parts. To add a WbApplication window to a VisualAge canvas, use the Options menu, Add Part submenu within VisualAge's Composition Editor. When dropped on the canvas, the window will appear as an icon representing a wrappered part.

New with V4.5, WindowBuilder Pro windows may be embedded within VisualAge windows as visual components (similar to CompositePanes within VisualSmalltalk) in addition to being used as standalone windows. WindowBuilder Pro windows placed on the Composition Editor canvas will appear as icons. If they are dragged and dropped onto a VisualAge window, they will show up as an embedded visual component. This makes WindowBuilder Pro the ideal environment for creating complex, reusable visual parts.

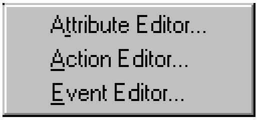
Once added to the Composition Editor's canvas, VisualAge connections may be made between the WbApplication window and other parts in the Composition Editor. Valid connection features for the window may be viewed from the Connect context sensitive menu for the part.

WindowBuilder Pro may be invoked directly from the VisualAge Composition Editor by double clicking on the WbApplication window's icon, which is analogous to opening the settings for a part within VisualAge. Changes saved through WindowBuilder Pro will be immediately available within the VisualAge editors. Changing a WbApplication subclass changes every occurrence of that class throughout the system.

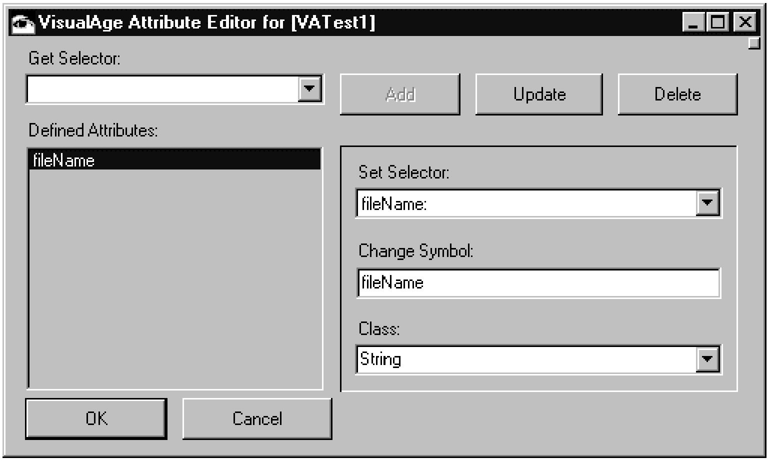
Defining VisualAge Connection Features In WindowBuilder Pro

Common connection features understood by all WindowBuilder Pro applications are defined by the WbApplication superclass. Connection features specific to a given WbApplication subclass must be defined through the VisualAge feature editors in WindowBuilder Pro.

Within WindowBuilder Pro, the VisualAge menu provides access to three editors which define the VisualAge connection features for a WbApplication subclass. The Attribute, Action, and Event editors respectively define VisualAge attributes, actions, and events for the class being edited. Each editor lists the non-inherited features already defined for the class, and allows the addition, deletion, or changing of features in the list.



Attribute Editor



Get Selector:

Defined Attributes: fileName

Set Selector: fileName:

Change Symbol: fileName

Class: String

The Attribute Editor allows the selection of a get selector from a list of non-inherited get selectors. The get selector corresponds to the attribute that will be exported by the class as a VisualAge attribute. The VisualAge attribute connection will use the chosen get selector to access the value of the exported attribute. Because attributes are accessed via selector, the list of possible attributes is limited to the possible get selectors for the class.

Once a get selector has been chosen, the editor attempts to find and display a matching set selector. The VisualAge attribute connection will use the chosen set selector to set the value of the exported attribute. The set selector should be left blank if the attribute being defined is a read-only attribute.

The change symbol defaults to the same name as the chosen get selector, but may be changed as necessary. The change symbol allows the window to signal an attribute change to other VisualAge parts that are interested in that attribute. The Attribute Editor allows the definition of the change symbol, but does not automatically generate code to signal attribute changes. Smalltalk code similar to the following fragment must be inserted into the set methods of the class being edited to signal the event represented by the change symbol:

```
self
   triggerCallback: #attributeName
   with: attributeValue.
```

Without such code, VisualAge parts that are connected to a WbApplication window’s attributes will never be signaled that the attributes have changed.

The attribute class defaults to Object, but may be changed as necessary. Only defined classes are accepted as valid. VisualAge uses the class of an attribute to determine if both ends of the attribute connection are compatible. Since Object is the ancestor of almost all classes, it provides the most generic compatibility. The attribute class may be changed to fine tune attribute compatibility.

The buttons on the Attribute Editor function as follows:

Add Adds the currently selected values to the list.

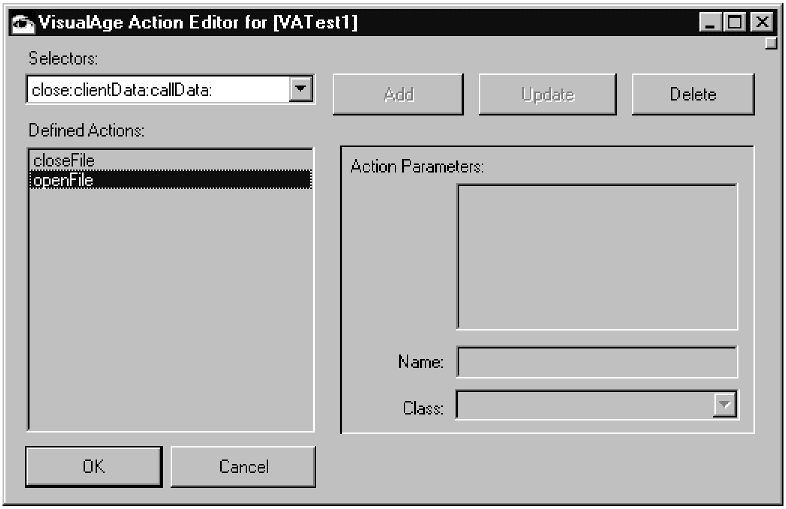
Delete Deletes the currently selected items from the list.

Update Updates the selected list item with any changed values.

OK Generates the class method containing the VisualAge attribute definitions (attributeSpecs) for the WindowBuilder application being edited and closes the window.

Cancel Closes the window.

Action Editor



VisualAge Action Editor for [VATest1]

Selectors: close:clientData:callData:

Defined Actions:
closeFile
openFile

The Action Editor allows the selection of a selector from a list of non-inherited selectors. The selector corresponds to the action that will be exported by the WbApplication subclass as a VisualAge action. The VisualAge action connection will use the chosen selector to invoke the exported action. Because actions are accessed via selector, the list of possible actions is limited to the non-inherited selectors for the class.

Once a selector has been chosen, the editor displays any parameters for that selector as appropriate. VisualAge uses the parameter information to provide a connection point for each parameter. VisualAge shows the parameter name on the context sensitive Connect menu for the action, and the parameter class is used to determine the compatibility between the parameter and the object it is connected with. The parameter name defaults to anObject, but may be changed as appropriate. The parameter class defaults to Object, providing the most generic compatibility. Only defined classes are accepted as valid.

The buttons on the Action Editor function as follows:

Add Adds the currently selected values to the list.

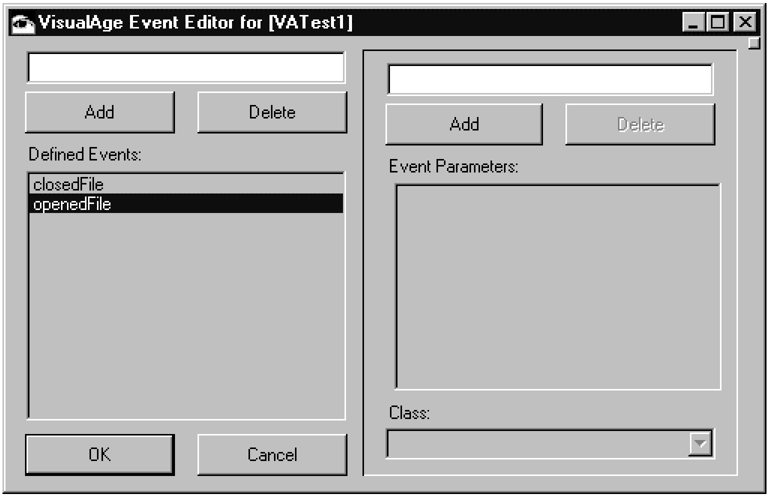
Delete Deletes the currently selected items from the list.

Update Updates the selected list item with any changed values.

OK Generates the class method containing the VisualAge action definitions (actionSpecs) for the WindowBuilder application being edited and closes the window.

Cancel Closes the window.

Event Editor



closedFile
openedFile

The Event Editor allows the entry and update of events and their respective parameters. Since events may not necessarily correspond to any selector, the editor does not present a list of selectors to choose from. Instead, it allows the user to enter events on the left, and enter any parameters for those events on the right. The VisualAge event connection will use the entered event name as the exported event.

Once an event has been entered, the parameters for that event may be altered. VisualAge uses the parameter information to provide a connection point for each parameter. VisualAge shows the parameter name on the context sensitive Connect menu for the event, and the parameter class is used to determine the compatibility between the parameter and the object it is connected with. The parameter name defaults to anObject, but may be changed as appropriate. The parameter class defaults to Object, providing the most generic compatibility. Only defined classes are accepted as valid.

Events may be signaled arbitrarily from pre-existing methods. Although WindowBuilder Pro generates the connection points for the events, it cannot generate the code necessary to signal the events. Smalltalk code similar to the following fragment must be inserted into the methods of the class being edited to signal the events defined in the Event Editor:

```
self
   triggerCallback: #eventName
   with: parameter1
   with: parameter2
   with: parameter3.
```

or:

```
self
   triggerCallback: #eventName
   withArguments: parameterValueArray.
```

Without such code, VisualAge parts will be able to connect to the events defined for a WbApplication window but those events will never be signaled.

The buttons on the Event Editor function as follows:

Add Adds the entered event or parameter to its respective list.

Delete Deletes the entered event or parameter from its respective list.

OK Generates the class method containing the VisualAge event definitions (eventSpecs) for the WindowBuilder application being edited and closes the window.

Cancel Closes the window.

Integration Examples Using WindowBuilder Pro And VisualAge

The tests in WbProVisualAgeExamples (a subapplication of WbProRuntimeExamples) demonstrate defining a VisualAge interface for a WbApplication and then using that application as a part within VisualAge. To view an example of WindowBuilder Pro and VisualAge integration, launch the VisualAge Organizer and do the following:

1. Select the WbProRuntimeExamples application

2. Click to the left of the WbProRuntimeExamples icon to reveal the subapplications. You may then double click on any of the example parts

Edit IntegrationTest1 to view a VisualAge application with an embedded WindowBuilder application (VATest1). The connection points provided by VATest1 were defined by the WindowBuilder VisualAge Feature Editors. IntegerationTest1 demonstrates an action connection and an event connection. The following behavior may be observed when testing IntegerationTest1:

- Clicking the “Open” button opens VATest1 on the file listed in the “Filename” box (via the open button’s clicked event connected to VATest1’s openWidget action).
- Closing VATest1 also closes the IntegrationTest1 window (via VATest1’s closedWidget event connected to closeWidget action of the main window).

Preferred Connections

VisualAge uses the notion of preferred connections to mean those attributes, actions, and events that appear cascaded from the Connect context sensitive menu choice. Non-preferred connections are those connections that appear when More is selected from the Connect submenu. By default, all the attributes, actions, and events generated for WbApplication windows appear on the More connections menu. To define any or all of these attributes, actions, and events as preferred connections, create a class method similar to the following for the WbApplication subclasses. This method defines the preferred attributes, actions, and events that will cascade from the Connect submenu:

```
preferredConnectionFeatures
   “Answer an array of the attributes, actions, and
    events that will appear on the Connect submenu”

   ^#(‘attribute1’ ‘attribute2’ ‘attribute3’ ...
      ‘action1’ ‘action2’ ‘action3’ ...
      ‘event1’ ‘event2’ ‘event3’ ...).
```

Index