

## WindowBuilder Pro/V 3.1

P46-0208-00

*Software to Simplify Our Complex World*



**Copyright © 1999–2000 Cincom Systems, Inc. All rights reserved.**

**Copyright © 1999–2000 Seagull Systems, Inc. All rights reserved.**

**This product contains copyrighted third-party software.**

**Part Number: P46-0208-00**

**Software Release 3.2**

**This document is subject to change without notice.**

**RESTRICTED RIGHTS LEGEND:**

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

**Trademark acknowledgments:**

CINCOM, CINCOM SYSTEMS, and the Cincom logo are registered trademarks of Cincom Systems, Inc. Visual Smalltalk is a trademark of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. Microsoft Windows is a registered trademark of Microsoft, Inc. Win32 is a trademark of Microsoft, Inc. OS/2 is a registered trademark of IBM Corporation. Other product names mentioned herein are used for identification purposes only, and may be trademarks of their respective companies.

**The following copyright notices apply to software that accompanies this documentation:**

Visual Smalltalk is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1999–2000 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

**Cincom Systems, Inc.**

**55 Merchant Street**

**Cincinnati, Ohio 45246**

**Phone: (513) 612-2300**

**Fax: (513) 612-2000**

**World Wide Web: <http://www.cincom.com>**










































# Contents

<b>Chapter 1 Introduction.....</b>	<b>1</b>
What is WindowBuilder Pro? .....	1
What you should already know .....	1
History of WindowBuilder Pro .....	2
How this manual is organized.....	2
Typographic Conventions.....	3
Technical Support .....	3
<b>Chapter 2 Overview.....</b>	<b>5</b>
Starting WindowBuilder Pro.....	5
Creating a window-based application .....	6
Step 1. Design the User Interface.....	7
Step 2. Attach callbacks to the interface objects .....	9
<b>Chapter 3 Using WindowBuilder Pro.....</b>	<b>13</b>
Positioning and Sizing Windows.....	14
Positioning and Sizing Widgets .....	15
Operations on Multiple Widgets .....	17
Using the Grid .....	22
Setting the Tab Order for Widgets .....	23
Changing Fonts.....	27
Setting Colors.....	28
Styles.....	30
Reframing Widgets.....	31
Widget Morphing .....	34
Using Call Outs .....	35
Using the Event Manager.....	36
Mnemonic Redundancy Checking .....	39
Using the Scrapbook.....	39
Using Popup Widget Menus .....	41

<b>Chapter 4 Menus &amp; WindowPolicies.....</b>	<b>45</b>
Creating a Menubar .....	45
Creating WindowPolicies .....	48
Popup Menus on Widgets .....	49
<b>Chapter 5 Coding in WindowBuilder Pro .....</b>	<b>51</b>
WindowBuilder Pro and Smalltalk .....	51
Passing Arguments to Windows.....	52
Returning Values From a Dialog.....	53
Naming Widgets.....	54
Passing messages from one widget to another.....	55
Alternate window opening protocols.....	55
<b>Chapter 6 Example Application.....</b>	<b>57</b>
Designing the Interface.....	58
Attaching Callbacks .....	60
<b>Chapter 7 Model Objects.....</b>	<b>61</b>
The Model Wizard .....	61
The Model Object Editor .....	67
Primary Models .....	68
The Layout Wizard.....	69
<b>Chapter 8 Toolbar Reference.....</b>	<b>73</b>
<b>Chapter 9 Menu Reference .....</b>	<b>75</b>
Transcript.....	75
File.....	80
Edit .....	88
View (obsolete) .....	92
Attributes .....	93
Align .....	106
Size .....	108
Options.....	109
Scrapbook .....	120
Add .....	122
<b>Chapter 10 Widget Encyclopedia.....</b>	<b>125</b>

---

All Widgets .....	125
SubPane & ControlPane .....	129
 AnimationPane .....	143
 Button .....	146
 CheckBox .....	148
 CheckBoxGroup .....	150
 ButtonListBox .....	153
 ComboBox .....	159
 DrawnButton .....	164
 DropDownList .....	166
 EnhancedEntryField .....	170
 EntryField .....	178
 EntryFieldGroup .....	182
 GraphPane .....	184
 GroupBox .....	186
 Header .....	187
 ListBox .....	190
 ListPane .....	196
 ListView .....	200
 MultipleSelectListBox .....	205
 ProgressBar .....	212
 RadioButton .....	214

	RadioButtonGroup .....	216
	RichEdit .....	219
	ScrollBar .....	229
	SexPane .....	232
	SpinButton .....	234
	StaticBox .....	239
	StaticText .....	240
	StatusPane .....	242
	StatusWindow .....	245
	TabControl .....	247
	TextEdit .....	252
	TextPane .....	257
	ThreeStateButton .....	261
	TrackBar .....	264
	TreeView .....	269
	UpDown .....	273
	VideoPane .....	276
	WBStaticGraphic .....	277
	WBToolBar .....	279
<b>Chapter 11 Windows and Dialogs .....</b>		<b>287</b>
<b>Chapter 12 CompositePanels .....</b>		<b>295</b>

---

Creating Composite Panes .....	295
Styles.....	295
Nesting .....	295
Tab Order .....	296
Adding Events .....	296
<b>Chapter 13 Pool Managers.....</b>	<b>299</b>
General Menu Definition .....	300
Pool.....	300
Category .....	300
Active.....	302
Key.....	302
Option .....	303
Bitmap Manager.....	304
Pool Menu .....	306
Category Menu .....	306
Key Menu.....	306
Option Menu .....	309
Bitmap Menu.....	309
Font Manager .....	312
Pool Menu .....	313
Category Menu .....	313
Option Menu .....	314
NLS Manager.....	315
Pool Menu .....	316
Category Menu .....	317
Programmatic Interface .....	318
Pool Manager Programmatic Assess .....	318
Pool Programmatic Assess.....	321
<b>Chapter 14 The Cookbook .....</b>	<b>325</b>
Widgets .....	325
Accessing a widget programatically.....	325
Changing a widget's size and position programatically.....	326
Changing a widget's font .....	327
Changing a widget's color .....	327
Changing a widget's label or contents.....	328
Hiding and showing widgets .....	328



Enabling and disabling widgets.....	329
Setting focus to a widget programatically .....	329
Forcing a widget to redraw itself.....	330
Windows .....	330
Opening a window.....	330
Opening a window floating above another .....	330
Opening a window as an MDI parent or child (windows only).....	331
Hiding and showing windows .....	331
Bringing a window to the front of other windows .....	332
Changing a window's size and position programatically .....	332
Changing a window's backcolor .....	333
Enabling and disabling windows.....	333
Changing a window's label.....	334
Adding a widget to a window dynamically .....	334
Adding a timer to a window.....	335
Minimizing a window to an icon.....	335
Restoring a window from an icon .....	336
Maximizing a window .....	336
Setting a window's icon.....	336
Finding the window under the pointer.....	337
Menus .....	337
Accessing a menu programatically.....	337
Changing a menu item's label.....	338
Enabling and disabling menu items.....	338
Checking and unchecking menu items .....	339
Removing menu items dynamically .....	339
Adding menu items dynamically.....	340
Dialogs.....	340
Closing unwanted dialogs .....	340
Displaying a message .....	341
Asking a yes/no question .....	341
Requesting a textual response .....	342
Potpourri .....	342
Dealing with impatient users.....	342
Debugging runtime errors .....	343
Fork with interrupts error.....	343
Runtime-less applications .....	344
Eliminating obsolete code.....	345

---

Migrating to the new event model.....	346
Adapting domain models to widgets .....	347
Naming panes, event handlers and other methods .....	349
Dynamic CompositePanels .....	352
<b>Appendix A Customizing WindowBuilder Pro .....</b>	<b>355</b>
The Graphic Object Framework.....	355
Graphic Object Naming .....	355
Setting a Widget's Contents.....	356
Setting a Widget's Initial Size .....	356
Setting the Minimum and Maximum Size.....	356
Working with Color.....	357
Setting the Default Font.....	357
Denying Input Focus.....	357
Adding Styles .....	358
Supporting the Layout Wizard.....	359
Enabling Morphing.....	359
Adding Pool Dictionaries.....	360
Drawing Your Widget .....	360
Other Customizations .....	361
Creating an Attribute Editor .....	362
Pre and Post Attribute Editing.....	365
Adding Widget Palette Icons .....	366
Using Add-In Modules .....	366
Adding Code Generation .....	371
<b>Appendix B PARTS Workbench Integration.....</b>	<b>373</b>
WB-PARTS Interface.....	373
WBMiniBrowserExample code .....	375
<b>Appendix C WindowBuilder Pro Tools Menu .....</b>	<b>381</b>
<b>Appendix D The Notifier Explained.....</b>	<b>387</b>
What is the Notifier? .....	387
Programming for OS/2 and Windows .....	387
Window Events .....	387
The Message Loop.....	388
The Window Function .....	388

An Overview of Event Processing in Smalltalk.....	389
Executing the Event Loop.....	390
Direct vs. Queued Window Messages.....	395
Synchronization Problems .....	396
Avoiding Synchronization Problems.....	398
Methods Relating to Queued Messages.....	398
The Notifier and Open Windows .....	398
Reinitializing the Notifier .....	399
Methods Relating to the List of Windows .....	399
The Notifier and the User Interface Process.....	400
Modal Windows .....	401
Trapping User Input Outside a Window.....	402
<b>Index.....</b>	<b>405</b>

# Acknowledgments

## Software Design and Development

Eric Clayberg, Dan Rubel

## Manual

Eric Clayberg, Ken Thompson, Dan Rubel

## Testing

Suman Goel, S. Sridhar, Solveig Viste, Max-Pieter Fränkel, Tom Murphy, Jeff Odell

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!

**Simon Archer, Wolfgang Baeck, Kent Beck, Robert Benson, Carter Blitch, Nik Boyd, James Chan, Ron Charron, Eric Clayberg, Karen Clayberg, Ken Cooper, Peter Day, Dina Fischer, Max-Pieter Fränkel, Amarjeet Garewal, Suman Goel, Dan Goldman, Robert Gurrieri, Steve Harris, Chris Hayes, Scott Herndon, Hal Hildebrand, Jim Howe, Ron Jeffries, Herb Kelsey, Darrow Kirkpatrick, Ed Klimas, Peter Knife, Kalpana Krishnaswami, Jasmin McCabe, Carmelo Montalbano, Tom Murphy, Jeff Odell, Glenn Osborne, Nick Payne, Ted Peters, Rene' Plourde, Roxie Roachat, Peter van Rooijen, Dan Rubel, Dan Shafer, Doug Shaker, Gordon Sheppard, Ed Shirk, Chuck Siemons, Harprett Singh, S. Sridhar, Mike Taylor, Ken Thompson, Steve Wessels, Scott Wlaschin, Kirk Wolf, Solveig Viste, Joseph Vollaro, Robert Yerex, Dave Zeleznik**



# 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 VisualSmalltalk 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 VisualSmalltalk
- familiarity with components and navigation of the operating system you are using

This manual assumes that you have a functional knowledge of VisualSmalltalk. 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 tutorial provided by the VisualSmalltalk documentation 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, Inc., took over the development of WindowBuilder in 1992. WindowBuilder Pro was developed by Objectshare Systems and appeared first for Digitalk Smalltalk/V in 1993 (and for IBM Smalltalk and VisualAge in late 1994). This version of WindowBuilder Pro for VisualSmalltalk includes all of the features of the previous version while adding new features not found anywhere else.

# How this manual is organized

Chapter 2 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 3 and 4 is a functional reference to the WindowBuilder Pro interface.

Chapter 5 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 6, you construct a simple application, using what you have learned in the first four chapters of this manual.

Chapter 7 discusses WindowBuilder Pro’s model object support.

Chapter 8 is an icon and toolbar reference.

Chapter 9 is a menu reference. Here you find information on every one of WindowBuilder Pro’s menu items.

Chapter 10 is a complete widget encyclopedia. All public protocols and supported events are listed here.

Chapter 11 covers windows and dialogs, and the window and dialog attribute editors.

Chapter 12 discusses how to create and write code for composite panes.

Chapter 13 covers WindowBuilder Pro's sophisticated pool management facilities including the Bitmap Manager, Font Manager and NLS Manager.

Finally, Chapter 14 is a comprehensive cookbook of common (and not so common) VisualSmalltalk GUI tasks.

## Typographic Conventions

This manual uses the following typographic conventions.

Example of convention	Used for
<code>printString</code>	Smalltalk code
<b>Style</b>	words that you type in
<code>C:\VSE31W\VDEVO.EXE</code>	file names
<i>input name</i>	placeholders for your input
<code>CNTRL+S</code>	key combinations

## Technical Support

Cincom Technical Support provides assistance by:

**email:** [supportweb@cincom.com](mailto:supportweb@cincom.com)

**web:** <http://supportweb.cincom.com/csc.html>

**telephone:** In North America call (800) 727-3525

Outside of North America, contact your local authorized reseller.

Support hours are 8:30 am to 5:00 pm, Eastern Time, Monday through Friday.



### General Note on Tech Support

When contacting a vendor for tech support, it is crucial to do your homework:

1. If you think you have discovered a bug, try to reproduce the bug in a *clean* image (totally clean if is a base image problem, or clean plus WindowBuilder Pro if you suspect a WindowBuilder Pro problem). Try to keep your example simple when doing this to remove as many other dependencies as possible. Most of the time people will discover their own errors and not need to call at all. In any case, doing so will help you clarify the problem in your own mind and help us help you all the quicker.
2. If you suspect a hardware problem (generally video driver related), try testing that by running your example under 640x480 256 color VGA, or on a co-worker's machine (with different hardware).
3. When you call us, have your complete walkback listing handy and be prepared to send us a small example to reproduce the problem (particularly if it can't be reproduced easily over the phone). Don't expect an immediate answer. If it is not immediately obvious, it might take us a little while to figure it out - hopefully not more than a day or two.

And above all else, be nice and try to remain objective (it might very well not be a bug in our product that is causing your problem). The nicer you are about it, the more likely we are to go out of our way to help you.

---

## Chapter 2 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 windows-based applications
- create a simple application

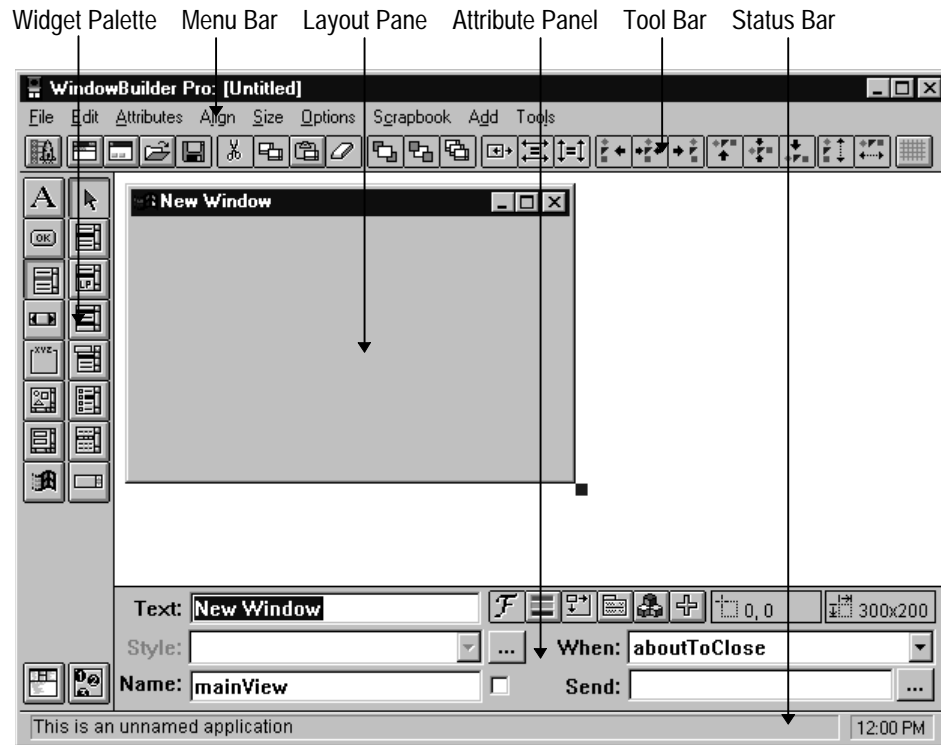
### Starting WindowBuilder Pro

The accompanying “Readme First” document contains instructions on how to install WindowBuilder Pro. After you have completed installation, start VisualSmalltalk.. You will notice that the installation procedure has added a new menu item, **WindowBuilder Pro**, to the Transcript menu bar.

**To start WindowBuilder Pro:**

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

The WindowBuilder Pro screen appears, as shown in Figure 2-1.



**Figure 2-1 The WindowBuilder Pro main screen.**

<b>Toolbar</b>	Displays icons representing shortcuts for menu commands.
<b>Widget Palette</b>	Displays icons representing the widgets available for placement in a window. You can also access these widgets from the Add menu.
<b>Layout Pane</b>	This is the work area in which you build your windows.
<b>Menu Bar</b>	Groups commands and options under text headings.
<b>Attribute Panel</b>	Contains entry fields and list boxes that allow you to add behavior to your widgets.

## Creating a window-based application

Application development in WindowBuilder Pro comprises two steps:

- Design the user interface.
- Attach callbacks to the interface objects.

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

*Attaching callbacks* is the process of associating widgets with Smalltalk methods. Callbacks (also called *event handlers*) are methods that launch when certain operations are performed on window objects. For example, when a button is clicked, a callback associated with the action “click” 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 TextPane and a Button. In this application, clicking the Button will cause the message “Hello, world!” to appear in the TextPane.

## Step 1. Design the User Interface

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

You add widgets to the window by choosing 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 choose a widget by first clicking its category icon in the left column, and then clicking the icon in the right column that represents the type of widget that you want from that category. For example, to choose a check box, first click the Button icon in the left column. 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 choose the check box.

Remember that you can also choose 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.

### 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.
2. Or, from the Add menu, choose the menu corresponding to the widget category, then choose the type of widget that you want.
3. 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. Position the cross hair where you want the upper left corner of the widget to be.

4. Hold the left mouse button down and drag the cross hair to draw the widget. Release the mouse button when the widget is the size you want.  
Or, position the cross hair and click the mouse button. This sets the widget to its default size.

The widget appears in the window, surrounded by four small black rectangles (handles).

**To resize a widget:**

- Point to one of the handles, and drag it until the widget is the size you want.
- If you want to resize in a horizontal or vertical direction only, hold down the **SHIFT** key, then drag in the desired direction.

**To move a widget:**

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

**To name a widget:**

- Select the Name entry field in the Attribute panel.
- Type the name you want to give to the widget. This name is used only as a code reference; it is not visible to the application user.

Some widgets, such as Buttons and StaticText widgets, can display text. You add text to a widget by using the Text entry field.

**To add text to a widget:**

- Select the Text entry field.
- Type the text you want to display in the widget.

## Designing the “Hello, World” Application

Now you have the procedures necessary to add widgets to the example application. You’ll use a Button and an EntryField widget. Place the widgets in a blank window, and add names and text to them as follows:

Widget	Name:	Text:
Button	helloButton	Push Me
EntryField	greetingField	(none)

When you are done, the screen should look similar to Figure 2-2.



Figure 2-2 Window for the “Hello, World!” application.

## Step 2. Attach callbacks to the interface objects

You attach callbacks to the currently selected widget by editing the Events field on the Attribute panel. The When: combobox and Send: entry field are the WindowBuilder Pro interfaces to the VisualSmalltalk. `when:send:to:` method. The events listed in the When: combobox are the events to which the window’s instance of ViewManager or ApplicationCoordinator can respond. You select an event from the list, and tell the window how to respond to the event by entering a method name in the Send: entry field. For example, if you want a message to display in a text pane when a button is clicked, you select the clicked event from the When: combobox, and enter a method name in the Send: entry field. In the Class Hierarchy Browser, add the code to display a message in a text pane. The window then knows that it is to display a message in a text pane when the button is clicked. This is how you add behavior to widgets.

Note that WindowBuilder Pro also provides a more advance Event Manager interface that allows you to add multiple callbacks for a single event. It can be accessed via the “...” button next to the Send: entry field or via the Events command in the Attributes menu. The Event Manager is discussed later on in this manual.

### To attach callbacks to widgets:

1. Select the widget for which you want to write the callback. Notice that the type of widget and its name or contents (if any) is displayed on the status bar.
2. Click the When: combobox at the bottom of the screen. Select the event to which you want the viewmanager to respond.
3. Click the Send: entry field. Type the name of the method that you want to execute in response to the When: event.

4. Repeat Steps 1-3 for the other widgets in the window that require callbacks.
5. Choose Save from the File menu. The first time you save the window, a dialog will prompt you to enter a name for a subclass of ViewManager or ApplicationCoordinator. This subclass will contain the methods for the objects in the user interface. Type in a class name, and click OK.
6. Click the Class Hierarchy Browser button in the lower left of the screen. A browser for the subclass that you have created appears. You see method names corresponding to all the entries you made for events in the Send: entry field. These methods are empty; you must write the code to handle the events.
7. Fill in the code for the methods, and save the class.

For the “Hello, World!” example, click the Button. Notice that the When: combobox displays **clicked** (the default event for Buttons). This is the event to which you want the window to respond, so you do not need to make any changes. Click the Send: field, and type **sayHello**. Save the “Hello, World!” window as a ViewManager subclass named **HelloWorld**. Click the CHB icon, and enter the code for the method, as shown in Figure 2-3. Save the method, and close the CHB.

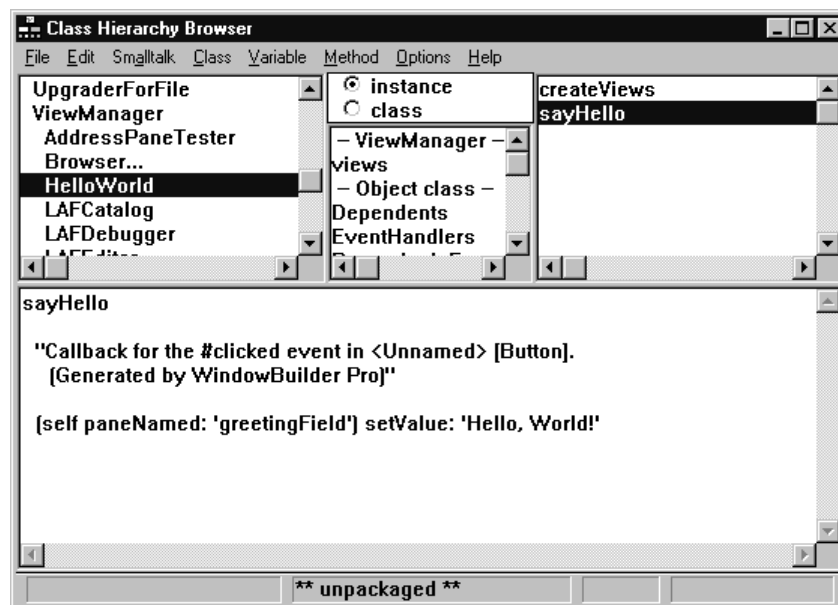


Figure 2-3 sayHello method.

#### To test an Application:

- Do one of the following:
- Click the Test Window icon.

- From the Edit menu, choose Test Window.
- Press CTRL+T.

To test the “Hello, World!” application, click the Push Me button. “Hello, World!” displays in the EntryField, as shown in Figure 2-4.



**Figure 2-4** “Hello, World!” application.

Congratulations! You have completed your first WindowBuilder Pro application. Although this is a very small example, you can see how WindowBuilder Pro simplifies writing windows-based applications in VisualSmalltalk.. You also have an understanding of the general application-creation process.

The following three chapters provide a reference to all of WindowBuilder Pro’s functions. A more sophisticated example application is presented in Chapter 6.





## Chapter 3 Using WindowBuilder Pro

In Chapter 2, 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 WindowBuilder 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
- Performing operations on multiple widgets
- Using the grid
- Setting the order for accessing widgets with the Tab key
- 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 Event Manager
- Mnemonic Redundancy Checking
- Using the Scrapbook
- Using Popup Widget Menus

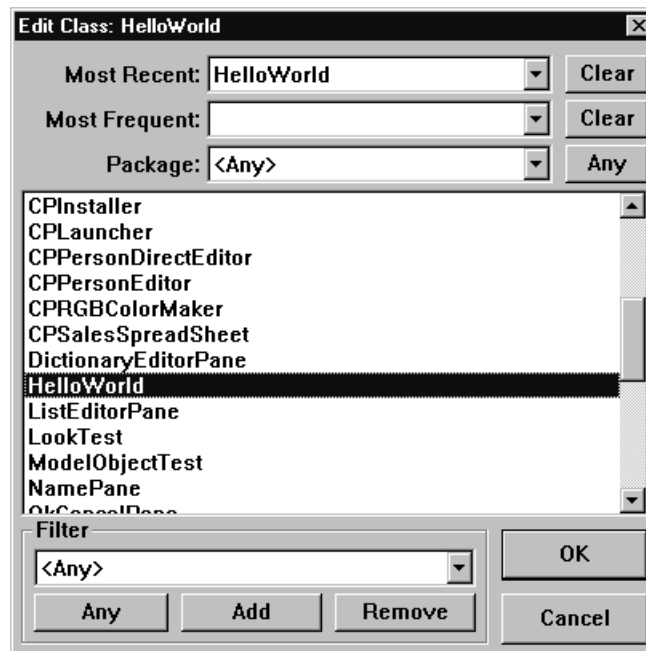
### Editing Windows

You can easily edit any windows that you have created and saved with WindowBuilder Pro. WindowBuilder Pro generates all window definitions as subclasses of `ViewManager` or `ApplicationCoordinator`.

#### To edit an existing window:

1. Choose Open from the File menu.  
Or, choose Edit Window... from the WindowBuilder Pro menu on the system Transcript.
2. The Edit Class Dialog appears, as shown in Figure 3-1.

3. Select a window to edit.

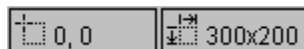


**Figure 3-1** Edit Class Dialog.

By default, the list in this dialog contains only those windows built by WindowBuilder Pro. The classes are listed alphabetically. Handy package/application and class type filters are provided to help you look at just the classes that you wish. The package/application filter is only available if you are using Team/V, ENVY/Developer or XoteryX. As a convenience, WindowBuilder Pro keeps track of the most recently accessed and most frequently accessed classes.

## Positioning and Sizing Windows

Two buttons are used to position and resize both windows and widgets. They are located on the Attribute Panel, as shown in Figure 3-2.



**Figure 3-2** Size and Position buttons.

To set the initial window position:

1. Select the window by clicking anywhere in the window other than on a widget.
2. Click the Position button. A rectangle that is the same size as the window appears.

3. Use the mouse to position the rectangle where you want the window to appear.
4. Click the mouse button when you are done. The rectangle disappears. The initial position of the window is set.

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

**To set the size of a window:**

1. Select the window by clicking anywhere in the window other than on a widget.
2. Click the Size button. The New Window size dialog appears.
3. Type a point representing the desired width and height of the window.
4. Click OK to close the dialog. The window resizes immediately.

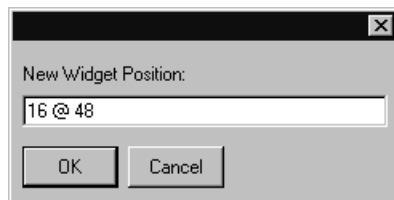
The Position and Size icons can also be used to position and resize selected widgets.

## 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 buttons shown in Figure 3-2. (You can also use these buttons to position and size windows. See the section, “Positioning and Sizing Windows.”)

**To position a widget:**

1. Select the widget that you want to position.
2. Click the Position button, or choose Set Widget Position from the Size menu. A dialog appears as shown in Figure 3-3, prompting you for the coordinates of the upper left corner of the widget.
3. Type the new coordinates.
4. Click OK. The widget immediately moves to the new position.

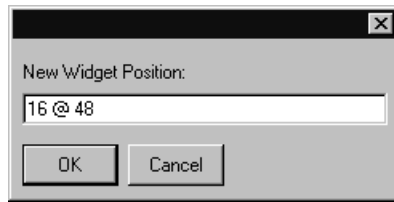


**Figure 3-3** New Widget Size Dialog.

**To size a widget:**

1. Select the widget that you want to size.

2. Click the Size button, or choose Set Widget Size from the Size menu. A dialog appears as shown in Figure 3-4, prompting you for the size of the widget.
3. Type the new coordinates.
4. Click OK. The widget immediately resizes.



**Figure 3-4** New Widget Position Dialog.

Remember that the size of the widget includes the widget's borders (if any). The sizes of the border and title bar are set in the Windows Desktop.

### 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 Align menu, but are more easily made using their accelerator keys (for information on accelerator keys, see Chapter 4, "Creating 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 move.

Or, choose Size By Pixel from the Align menu. A submenu will appear, displaying a menu item for each direction. Choose the direction toward which you want to resize the widget.

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

#### To resize a widget in 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.
3. Or, choose Size By Pixel from the Align menu. A submenu will appear, displaying a menu item for each direction. 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 StaticText or Button widget, the size would be just large enough to fully contain the widget's label.

### To Autosize a widget:

1. Select the widget you want to automatically size.
2. In the Text entry field, type the text label as you want it displayed.
3. 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 StaticText and Button.

## Operations on Multiple Widgets

It is useful to be able to perform simple editing operations, such as moving and deleting, on groups of widgets. This section covers how to create a group of widgets, and 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 from the palette.

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

- Place the widget in the window 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 three ways to select a group of widgets.

- The Select menu options: Select All, Select All In Same Class, Select All In Same Hierarchy.
- The Rubberband method (also known as the marquee selection method).
- The SHIFT-select Method.

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 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. In these operations, the order in which the widgets are selected is not important.

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, select 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.

## Replicating Widget Sizes

Often you need to make widgets the same size. You can either select one widget and make multiple copies by using the Copy command, 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:

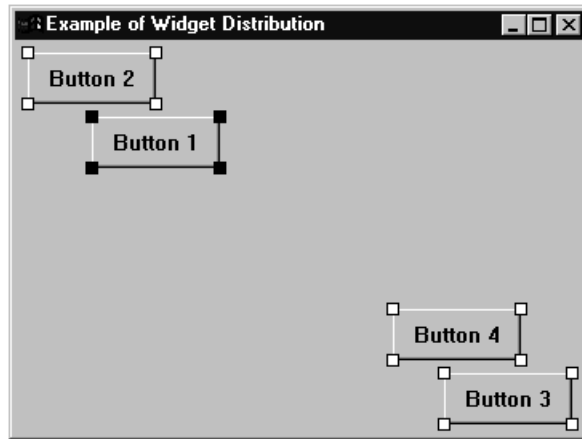
1. Position the two endpoint widgets where you want the ends of the row or column to be. Leave one of the endpoint widgets selected.
2. SHIFT-select the rest of the widgets to be distributed. Make sure to select the other endpoint last.
3. Choose Distribute from the Align menu, or click the Distribute button. A submenu appears, prompting you for the direction of distribution (horizontal or vertical).
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.

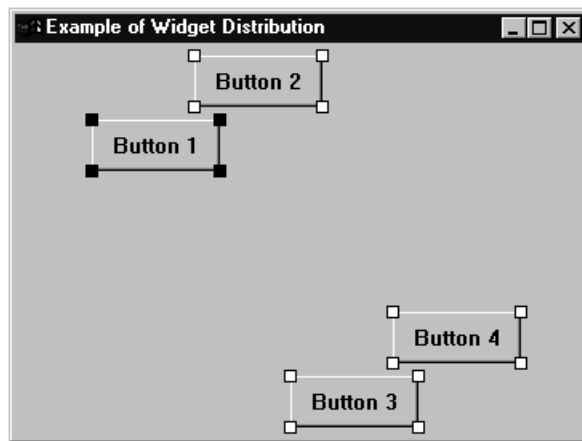
If the ALT key is held down while performing a horizontal or vertical widget distribution, the widgets will be distributed based on their relative position rather than the order in which they were selected.



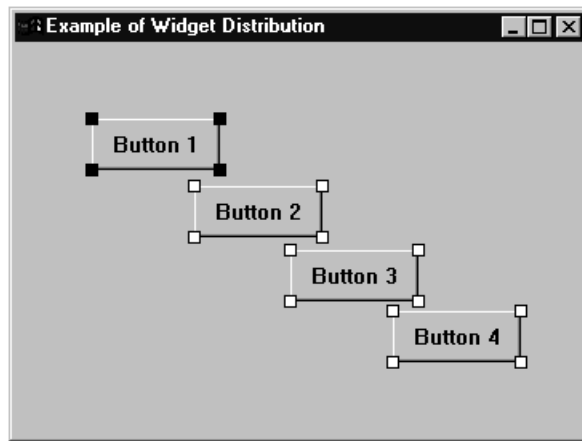
In the example shown in Figure 3-5, the four buttons have been selected in numeric order with Button 1 first, and Button 4 last. Selecting Distribute distributes them, resulting in the window shown in Figure 3-7.



**Figure 3-5** Buttons, before horizontal Distribute operation. Button 1 and Button 4 are in desired vertical position. Button 1 has been selected first, Button 4 has been selected last.



**Figure 3-6** Buttons, after the horizontal Distribute operation.



**Figure 3-7** Final position of buttons, after vertical 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.

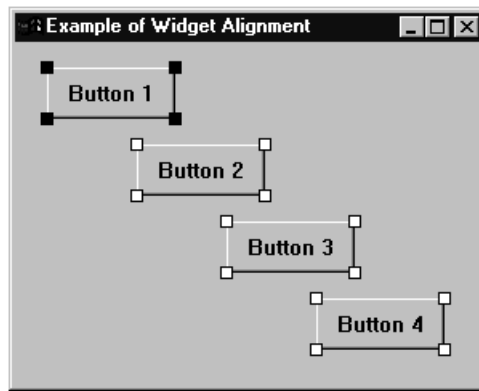


Figure 3-8 Buttons, ready to be aligned. Button 1 is the model widget.

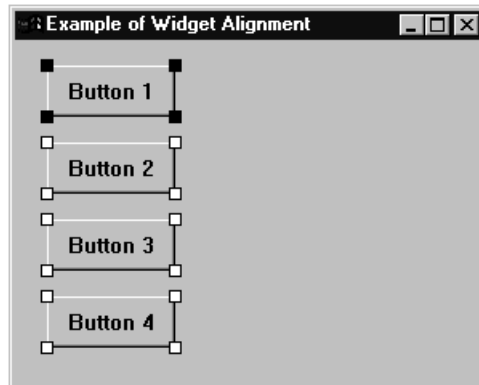


Figure 3-9 Buttons after left alignment.

## Using the Grid

In addition to its set of alignment tools, WindowBuilder Pro also offers a snap-to-grid feature for aligning a group of widgets. When widgets are moved, sized, or placed, they will 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 Property Editor. 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:

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 display the grid:

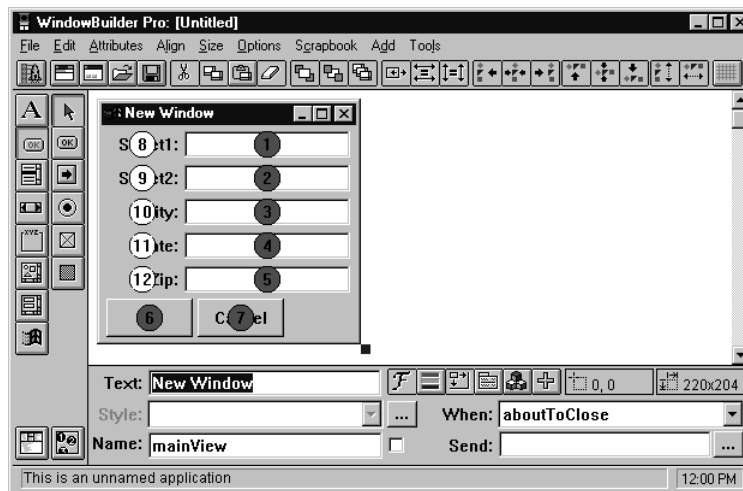
- Choose Draw Grid from the Options menu, or click the Grid button.

## 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 Z-order:

- Choose Show Z-Order from the Options menu. A round, color-coded number appears on each widget as shown in Figure 3-10.



**Figure 3-10** Show Z-Order Display.

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 tab stops.

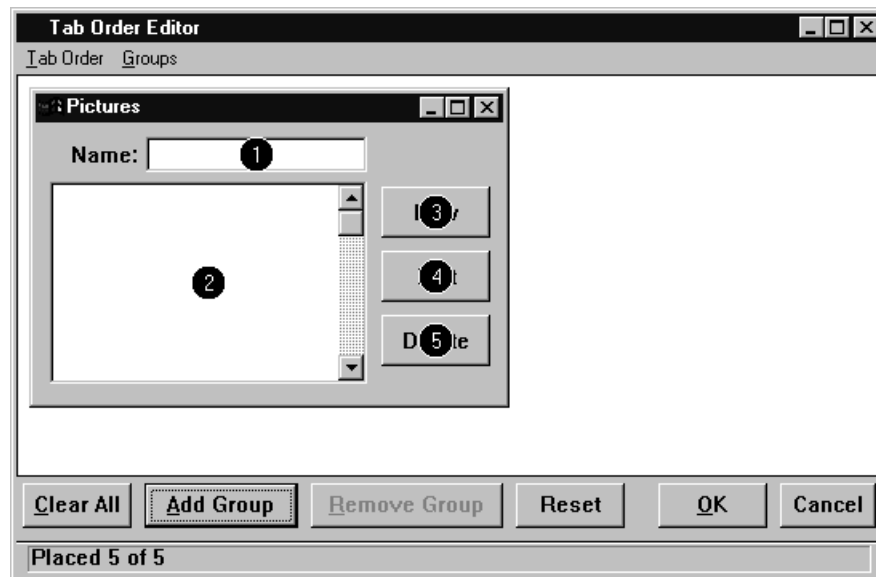
**White** indicates widgets that are not tab stops (for example, labels and separators).

To set the tab order:

- From the Edit menu, choose Edit Tabbing/Groups.

Or, click the Tab Order Editor button.

The Tab Order Editor appears as shown in Figure 3-11. This editor displays any widgets you've laid out with WindowBuilder Pro. If you have already entered a tab order for the widgets, numeric labels representing this order are displayed on each widget. If you have not yet set the tab order, no labels are displayed on the widgets.



**Figure 3-11** Tab Order Editor.

Click the widgets in the order in which you want them to be selected with the TAB key. A numeric label appears on each widget when you click it, representing its position in the tab order. If you click a widget that does not make use of the input focus (such as aStaticText widget), the system beeps, and no number will appear on the widget.

If you make a mistake, simple ALT click on a widget to remove its tab stop. To swap two widgets in the tab order, click on the first widget and drag its tab stop number to the second widget. This drag drop capability makes it easy to rearrange the tab order without having to start over again.

Choose OK to dismiss the Tab Order Editor dialog. Note that the Tab Order Editor is a modeless window and may be kept open simultaneously with WindowBuilder Pro. The Tab Order Editor will automatically refresh itself whenever it is re-activated.

The Tab Order Editor also provides a number of advanced functions for automatically establishing default tab orders for you.

**To establish a default tab order:**

- In the Tab Order Editor, select one of the Smart Set options from the Tab order menu. The default tab order may be establish in either column or row major order (or based on the current z-order in the window). These options use an intelligent algorithm that will automatically identify nested widget groups.

The Tab Order Editor also provides options for clearing all of the tabs or for reversing the tab order for the entire window or for any of the tab groups.

## Creating Tab Groups

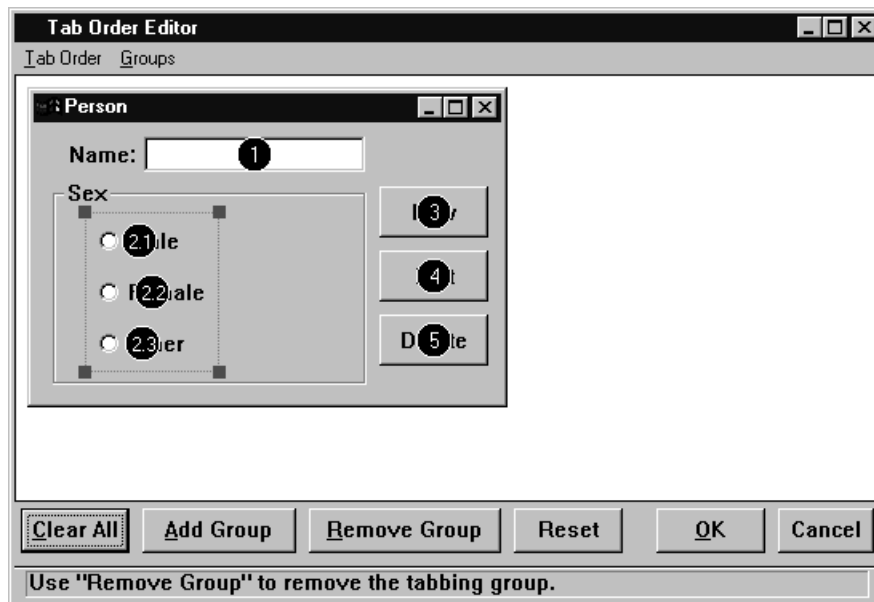
Tab groups are also supported. Most commonly used for groups of radio buttons, this construct is used to delineate a group of controls that should be grouped together for tabbing purposes. When you tab into a group, you can then use the arrow keys to move from one element to another within the group. Groups are also used to denote a collection of radio buttons that should be treated as a unit (this is especially useful for auto radio buttons, which turn themselves on and off within their group to ensure that only one button is selected at a time).

To support the grouping feature, the Tab Order/Grouping editor provides a mechanism for specifying the different groups within a window.

**To create a tab group:**

1. In the Tab Order Editor, press the Add Group button.
2. Rubberband-select the widgets that you want to group. A rectangular box appears around them, indicating that they're now a group. In addition, the previous tab order will be readjusted, to indicate that these components are part of a group.

In Figure 3-12, a group of radio buttons has been grouped. Notice that the numeric labels on the radio buttons indicate that the group is positioned second in the overall tab order. The numbers to the right of the decimal points indicate the order in which the buttons are selected within their group.



**Figure 3-12** Tab Order Editor with a group selected.

To remove an existing tab group:

1. Press the Remove Group button.
2. Click the group you wish to remove. The rectangle around the previously associated widgets is removed, indicating the group has been deleted. The numbering also changes to reflect the deletion.

To re-size an existing tab group:

1. Click on an existing tab group. Red sizing handles will appear on its four corners.
2. Drag one of the sizing handles in order to expand or contract the tab group to encompass a different number of selections.

To automatically establish tab groups based on the window layout:

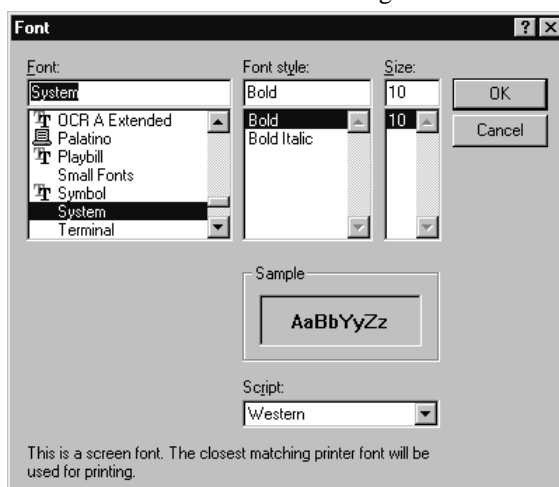
- Select the Create All option from the Group menu. The Tab Group Editor will establish tab groups based on the layout of the window (e.g., via the existence of GroupBoxes and similar grouping constructs). The Include Outer option in the Group menu determines whether the system will create a group for any buttons that are outside of any GroupBox.

## Changing Fonts

By default, the standard system font is used for all widgets. WindowBuilder Pro makes the standard Windows Font dialog available to you to change the font of a widget, or group of widgets.

### 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 system font dialog will appear, as shown in Figure 3-13.
3. Specify the Font, Style and Size of the desired font.
4. Click OK to confirm the change.



**Figure 3-13 Standard Font Editor.**

If you are using WindowBuilder Pro's advanced font pool management capabilities, you will see the font dialog shown in Figure 3-14. This dialog allows you to select a pre-defined named font from an existing font pool. The standard font editor may be launched from this dialog by clicking on the Select button. If you are using font pool management and you want to select a font directly from the standard font dialog, you may do so by holding the ALT key down when clicking the Font button.





Figure 3-14 Font Editor.

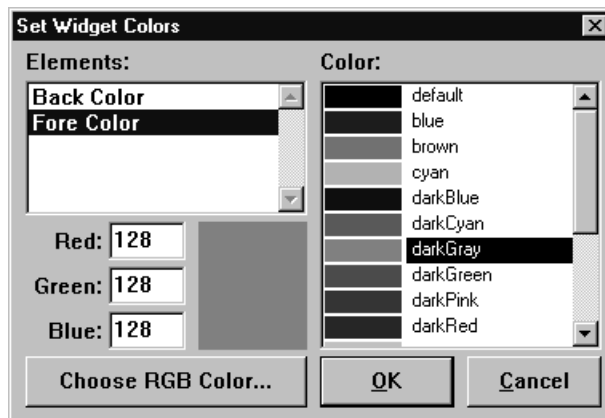
The Font button is disabled if the font is not a settable attribute for the selected widget.

## Setting Colors

Each type of widget 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:

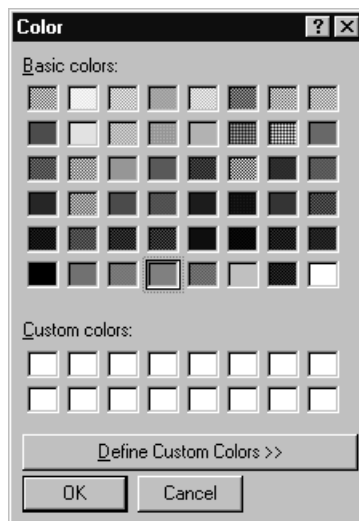
1. Select the widget or widgets whose colors you wish to change.
2. Click the Color button, or choose Color from the Attributes menu. The Color dialog appears, as shown in Figure 3-15.
3. Select the color attribute that you wish to change (generally just “Back Color” and “Fore Color”). When a color attribute 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 left.
4. Either select the desired color from the color list, or use the RGB entry fields to set the red, green and blue values of the desired color.
5. Click OK to confirm the color choice.



**Figure 3-15 Color Editor.**

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.

The system color editor can be invoked by clicking on the Choose RGB Color button. It is shown in Figure 3-16



**Figure 3-16 System Color Editor.**

## Styles

Many widgets have multiple styles from which you can choose. A StaticText widget, for example, can be left, right, or center justified; Buttons can be regular or default buttons. The Style combo box displays the styles available for the selected widget. If the widget in which you're interested has multiple styles, this combo box will contain several choices.

### To change the style:

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

The Style Editor is useful for setting a widget or window's boolean properties or operating system specific style attributes. Choose the Styles command from the Attribute menu or click on the "..." button next to the style combo box to open the Style Editor shown in Figure 3-17. Once open, you will see a button list box containing the available styles for the currently selected widget.

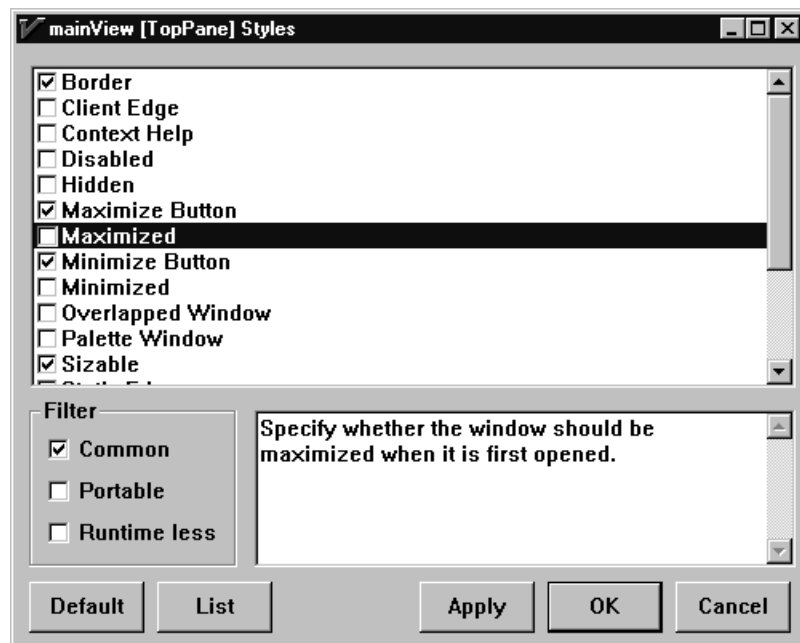


Figure 3-17 Style Editor.

Below that list is a Filter group box containing three checkboxes. These checkboxes, when selected, filter the list of styles to include only Common styles (as in commonly used), Portable styles, and/or Runtime less styles. For example, if you select Portable and Runtime less but Common is not checked, then the list will display only those styles that are both Portable *and* Runtime less. To the right of the Filter group box appears the description of the currently selected style.

Along the bottom of the window are five buttons. The Default button sets the values of currently visible styles to be equal to the styles of a newly created widget. The List button opens a text window displaying a list of all currently visible style and a description for each style. If you hold down the Control key while clicking on the List button, then the text window will contain a list of styles for *all* widgets (Window subclasses). The Apply button applies the currently selected style values to the currently selected widget. If more than one widget is selected, it will change the properties for selected widgets that have the same class as the primary selected widget. The OK button is the same as Apply, but also close the window. The Close/Cancel button closes the window without setting the currently selected widget's styles.

## 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 VisualSmalltalk., this is usually accomplished with a framing block; a block is passed in, with the rectangle of its parent. It is then up to you in this block to calculate the widget's rectangle based on its parent's rectangle. This is a flexible mechanism, but can be cumbersome to use.

### LayoutFrames

WindowBuilder Pro supports a special framing mechanism that automates most situations where you might need a framing block. This mechanism, called *LayoutFrames*, allows you to directly set the positions of the right, left, top, and bottom of a widget, relative to its enclosing window.

LayoutFrames are useful only for resizable windows. Since dialogs are not resizable, the Framing button is disabled while you are editing dialogs.

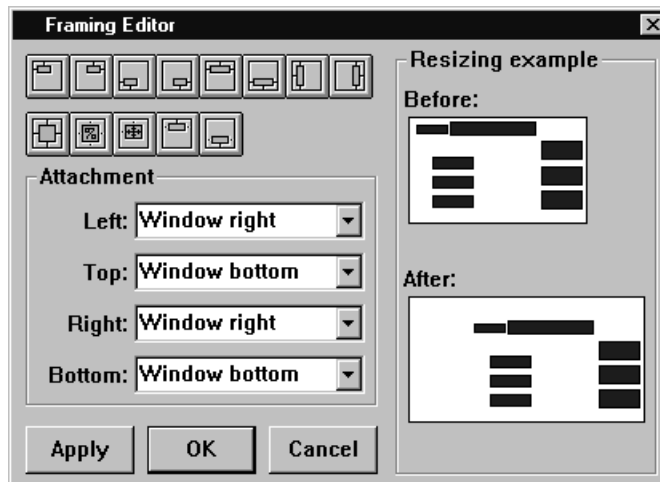
#### To set the framing specification:

1. Select the widget or widgets whose framing you wish to change.
2. Press the Framing button on the attribute panel, or choose the Framing command from the Attributes menu. The Framing Editor appears, as shown in Figure 3-18.
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

window's left side, the window's right side, the window's center, a fixed relative to its own right side (None), or relative (proportional) to its initial position on the window.

4. Click OK or Apply to confirm the attachment choices.

The four combo boxes are also editable and will accept fractions (less than one). This fraction represents some ratio of the window's width or height from which the widget's offset will be calculated. For example, left and right fractions of 1/2 will lock the widget relative to the horizontal center of the window.



**Figure 3-18 Framing Editor.**

The two diagrams on the right of the window illustrate the effect the framing specification you've set will have on the widgets when the window resizes. The top diagram represents the widgets in the window before the window has sized; the bottom diagram represents the widgets in the window after the window has sized.

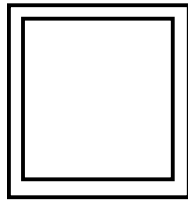
At least one side in both the vertical and horizontal directions must be specified. If, for example, a widget's left side is fixed relative to its own right side, the framing specification (attachment) for the widget's right side must be fixed somehow relative to its window. Otherwise, you'd have a pretty confused widget floating around!

As an alternative to fixing the left side relative 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.

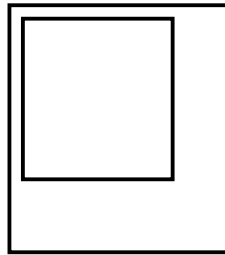
As mentioned above, this discussion holds true for all four sides of a widget. Since each side can be specified separately from the other sides, you can create many different

variations, covering most common resizing situations. To help illustrate the possibilities, we've provided the following examples:

Before

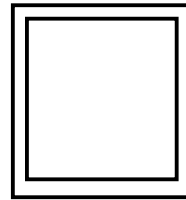


After

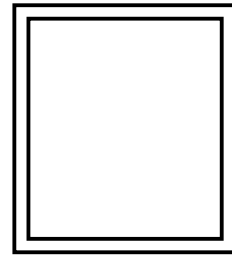


Left relative to window left, top relative to window top, right relative to widget left, bottom relative to widget top.

Before

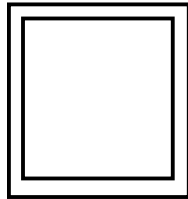


After

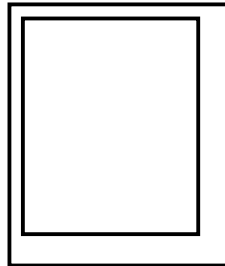


Left relative to window left, top relative to window top, right relative to window right, bottom relative to window bottom.

Before

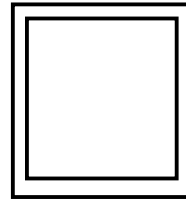


After

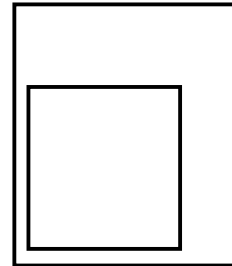


Left relative to window left, top relative to window top, right and bottom proportional.

Before

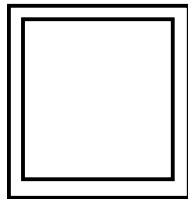


After

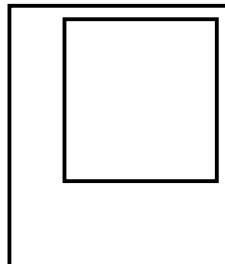


Left relative to window left, top relative to widget bottom, right relative to widget left, bottom relative to window bottom.

Before

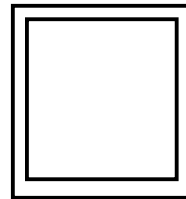


After

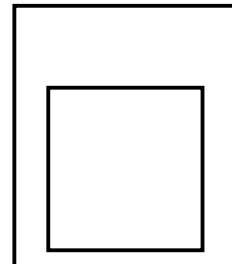


Left relative to widget right, top relative to window top, right relative to window right, bottom relative to widget top.

Before



After



Left relative to window center, top relative to widget bottom, right relative to window center, bottom relative to window bottom.

### Framing Examples.

**To set the framing specification for multiple widgets:**

1. Multiple-select the desired widgets.
2. Click the Framing button.

For more information on multiple selection, see [Selecting Multiple Widgets](#).

If the Update Outboards option is on, the Framing Editor may be left open. As widgets are selected in the main editor, their current settings will be reflected in the Framing Editor. They may then be changed, and those changes locked in, by clicking the Apply button.

**Framing Styles**

Many different framing combinations can be specified. However, you will probably only use a few of the possible combinations on a frequent basis. Recognizing this, the Framing Editor provides a fast path to the common attachment combinations, or *styles*. The toolbar at the top of the Framing Editor provide access to all of the most common framing 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 appropriate toolbar button.

## 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 `ListBox` instance could be converted into a `ComboBox` or `RadioButtonGroup` instance. 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 Edit menu.
3. Or, right-click, and select Morph from the popup menu as shown in Figure 3-19. This is the more desirable method, as it provides a list of similar widget types from which to choose (others may be accessed through the **Other...** command).
4. Select the new widget class from the list of widget types presented.
5. Click OK to morph the widget into the new type.

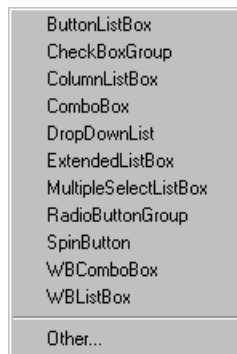


Figure 3-19 Sample Popup Morph Menu.

Be careful when morphing a widget into a radically different type. WindowBuilder Pro maps over any attributes the two have in common as well as callbacks for any shared events. You must be careful that the callbacks do, in fact, make sense for the new type. For example, a **needsContents** callback for a listbox would not be appropriate for an entryfield. 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 `createViews` method (which can cause problems with the VisualSmalltalk compiler). The Call Out Editor shown in Figure 3-20 allows you to have any widget or model object generated into its own method.

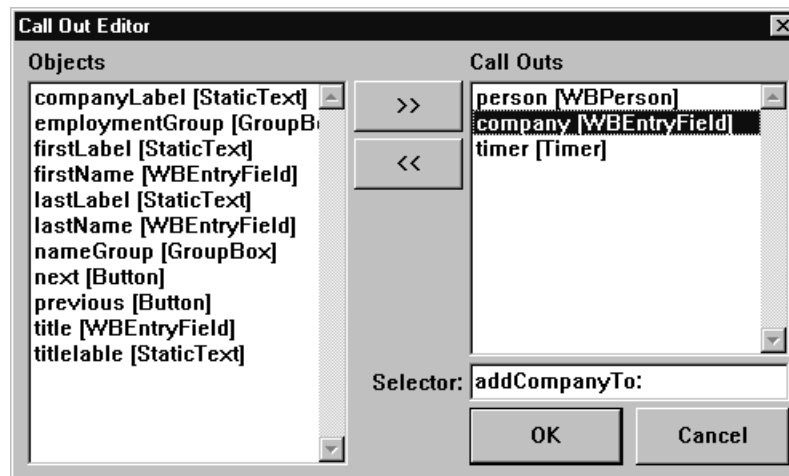


Figure 3-20 Call Out Editor.



**To create a call out:**

1. Open the Call Out Editor.
2. Select a widget or model object 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 a one-argument selector.
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 Event Manager

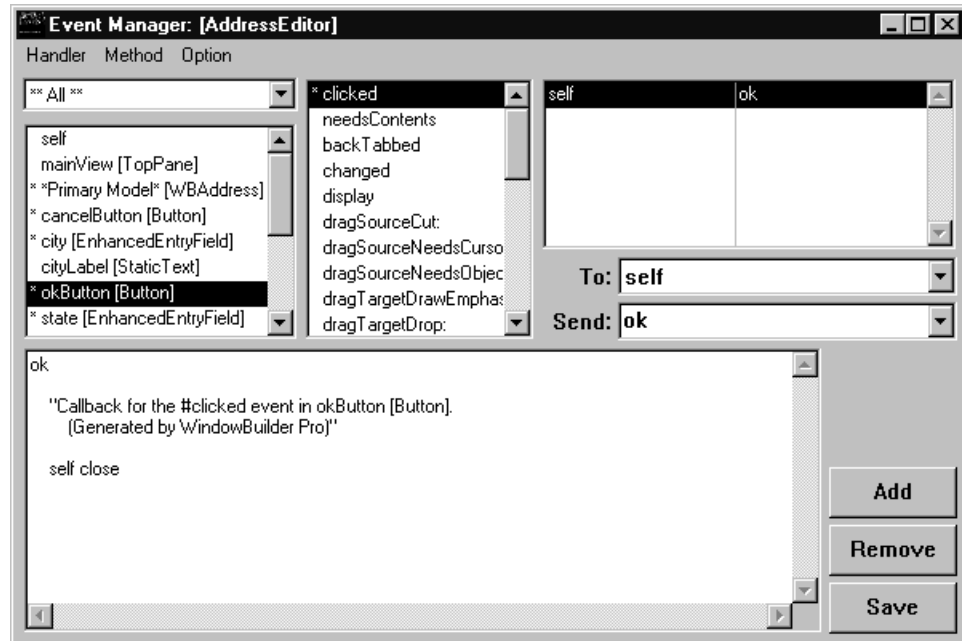
The Event Manager window creates, edits, and removes event links between objects. Any window, widget, model, or global that triggers events can be linked to a method of any other window, widget, model, or global. For example, using the Event Manager to create a link between the `#clicked:` event of a list box and a method called `#myListBoxClicked:`, when the user selects an item in that list box, the `#clicked:` event is triggered, and a message is sent to the `#myListBoxClicked:` method where processing occurs to handle the user’s selection. In addition, you can create and edit the event handlers (the `#myListBoxClicked:` method) in the Event Manager window.

In the Event Manager window shown in Figure 3-21, the object list (left top of the Event Manager window) displays the objects that trigger events. If the object has links to one or more of its events, then an asterisk appears preceding the name of the object. Directly above the object list is a combo box used for filtering the list of objects.

The event list (center top of the Event Manager window) contains the events for those objects selected in the object list. The events appear in alphabetical order.

The handler list (right top of the Event Manager window) contains the handlers defined for the objects selected in the object list and the events selected in the event list. If one object and one event is selected, then the handler list displays the target object (the object receiving the message) and the target selector (the message sent to the target object when the event is triggered). If multiple objects in the object list are selected, then the handler list displays the source object (the object triggering the event) in addition to the target object and the target selector. If multiple events are selected, then

the handler list displays the event being triggered in addition to the target object and the target selector.



**Figure 3-21 The Event Manager.**

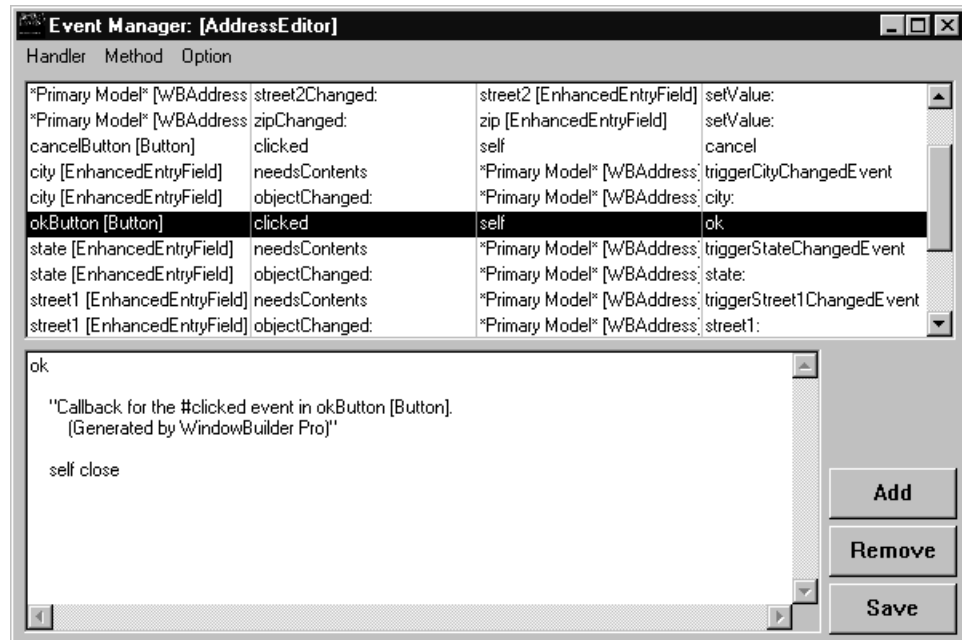
Below the handler list appears the target object drop down list and the target selector combo box. If no handler is selected in the handler list, then selecting a target object in the target object drop down list and entering a target selector in the target selector combo box will add a new handler to the handler list for the selected source object and event. If a handler is selected in the handler list, then selecting a target object or changing the target selector will change the target object and/or target selector for that selected handler.

To enter a new handler when no handlers are selected, select the target object and enter the target selector in the target object drop down list and target selector comb box respectively. To enter a new handler when one or more handlers are selected, select Add from the Handler menu, then select a target object and enter a target selector. To edit a handler, select the handler and then alter the target object and/or target selector. To remove a handler, select the handler then select Remove from the Handler menu.

If there is one source object selected and one event selected and multiple handlers appear in the handler list, then the order in which the handlers appear in the handler list is the order that the handlers are attached to a source object and event. If you wish to change the order, select a particular handler and use the Top, Up, Down, and Bottom

menu items in the Handler menu. Alternately, you may drag and drop the handlers using the mouse.

The Show All command in the Handler menu expands the handler list to fill the entire upper half of the Event Manager window and displays the source object, event, target object, and target selector for all defined handlers. This is shown in Figure 3-22.



**Figure 3-22** The Event Manager with expanded handler list.

The method source for the currently selected handler appears in the text edit area in the bottom half of the Event Manager window. If there is a method already defined for the target selector in the target object's class, then the source for that method will appear in the text edit area. If there is not a method already defined and the target object is 'self' then the source for a method stub will appear in the text edit area. You may edit and save method source in this text edit area.

The method text editor is also compatible with Cooper & Peters' **edIt**<sup>™</sup>, the programmer's editor for Smalltalk. **edIt** is a must-have product for doing serious code development in VisualSmalltalk.. With **edIt** installed, the Event Manager looks like Figure 3-23.

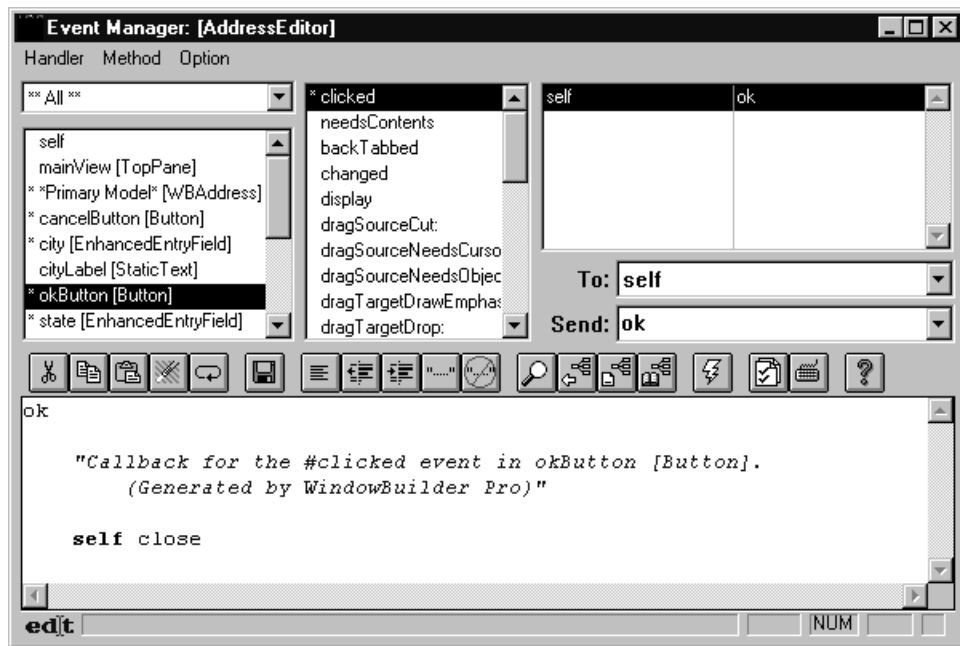


Figure 3-23 The Event Manager with edIt installed.

## Mnemonic Redundancy Checking

Associating a mnemonic key with a widget enables a user of an application to access the widget with the keyboard. (For a discussion of mnemonic keys, see Chapter 4, “Menus”). It is important not to assign a mnemonic key to more than one widget in the same window. When you assign a mnemonic key to a widget, WindowBuilder Pro automatically checks to see if the key is assigned to another widget in the window. If it is, WindowBuilder Pro alerts you.

When in the menubar editor, you’ll be alerted if you set the mnemonic of two menu items within the same menu to the same value.

## Using the Scrapbook

The Scrapbook provides a convenient place to store reusable components and standard layouts. It eliminates the need to create the same layout over and over again. For example, if more than one of your dialogs requires an “OK&Cancel” button combination, you can store this pair of widgets in the Scrapbook to reuse again. Scrapbooks may be saved to disk and merged together, allowing multiple developers to easily share layouts between them.

The Scrapbook is organized into chapters and pages. A page consists of one entry in the Scrapbook which has one or more items in it (for example, the “OK&Cancel” button combination would be one page). A chapter is basically a category. Chapters can be made up of one or multiple pages. You may have as many chapters as you like, and you may assign a page to multiple chapters.

#### To store widgets in the Scrapbook

1. Select the widget or widgets you want to store in the Scrapbook.
2. From the Scrapbook menu, choose Store.... The Store in Scrapbook Dialog appears, as shown in Figure 3-24.

The Page Name entryfield is used to record the name of the new page. The listbox below it shows all of the chapters that have been defined. New chapters may be easily added by clicking on the New Chapter button. You may select as many chapters as you like in which to store the new page.

**Note:** The *Quick Reference* chapter is special. Any pages you put in it will appear as cascaded menu items on the Scrapbook | Quick Reference menu.

3. Enter the page name that you want to assign to the widget group.
4. Select the chapter, or chapters, to which you want to assign the page.
5. Click Store to store the widgets in the scrapbook. Cancel will quit the dialog without storing the widgets.

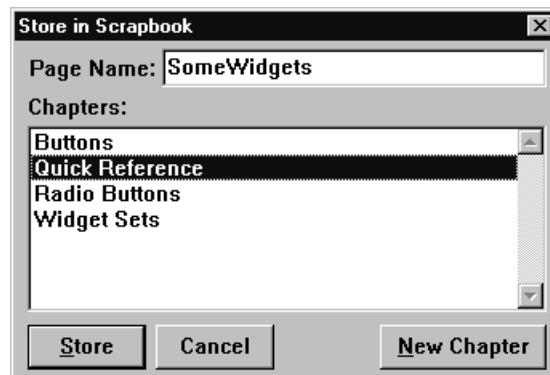


Figure 3-24 Store in Scrapbook Dialog.

#### To retrieve widgets from the Scrapbook:

1. From the Scrapbook menu, choose Retrieve.... The Retrieve from Scrapbook Dialog appears, as shown in Figure 3-25.

The Chapters listbox lists all of the defined chapters. The Pages listbox below it shows all of the pages that make up the chapter.

2. Select the chapter containing the category of widgets you want to retrieve. The name of the first page of the chapter and its widgets are displayed. You can scroll through the list of pages by using the scrollbar located at the bottom right of the dialog, or by clicking the names of the pages in the Pages list box.
3. Select the page that is assigned to the widgets you want to retrieve.
4. Click OK to retrieve the widgets, or Cancel to cancel the Retrieve operation. The dialog disappears. If you click OK, the cursor is loaded with the widgets.
5. Place the widgets at the desired position in your window.

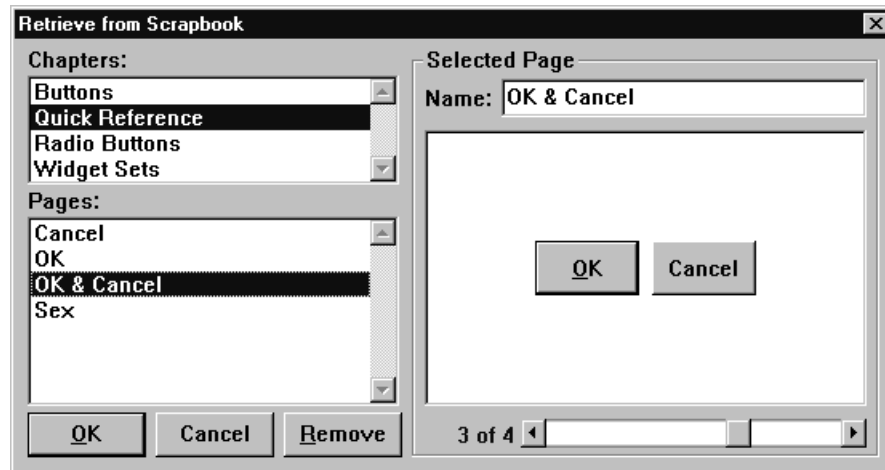


Figure 3-25 Retrieve from Scrapbook Dialog.

## 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 in Figure 3-26. This menu gives you quick access to setting the background color of the window, editing its menubar, the event manager, the style editor and its attribute editor. It also allows you to open a class browser on the window's class and gives you the option of saving the current window as the default template.



**Figure 3-26** Popup Window Menu.

If a single widget is selected, the widget popup menu appears as shown in Figure 3-27. This menu gives you quick access to the widget's font, color, framing, popup menu definition, events, styles 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 cascading list of similar widget types that the current widget may be morphed to.



**Figure 3-27** Popup Widget Menu.

If a multiple widgets are selected, the widget group popup menu appears as shown in Figure 3-28. This menu gives you quick access to the widgets' font, color, framing, and events. The selected widgets may be converted into a new CompositePane subclass via the Create Composite option. The widgets may be aligned using any of the alignment options. The menu also allows you to open class browsers on the classes of the selected widgets 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 cascading list of similar widget types that the current widgets may be morphed to.



**Figure 3-28** Popup Group Menu.





## Chapter 4 Menus & WindowPolicies

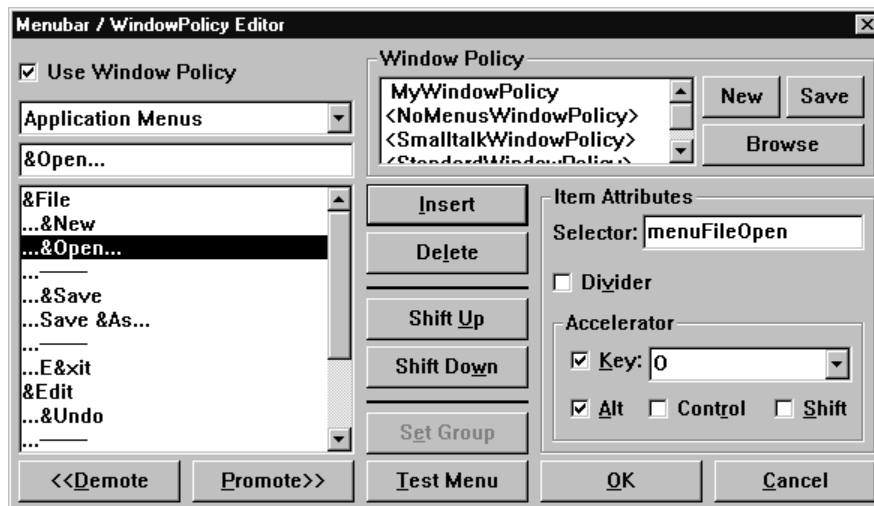
WindowBuilder Pro supports hierarchical menus on both widgets and windows. When you attach a menu to a widget, it is called a popup menu. A menu under the title bar of a window is called a menubar. The process of adding menus to widgets and windows is almost the same. This section discusses the procedures that you use to create menubars for windows. Menu definitions may be stored with the class itself or may be saved as separate WindowPolicy subclasses. WindowPolicies provide a convenient mechanism for sharing menu definitions between multiple windows. Issues regarding popup menus 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 menubar editor.

**To add a menu bar to a window:**

1. Select the title bar of the window, or a section of the window not covered by a widget.
2. Click the Menu bar button on the Attribute panel.  
Or, select the Menus command from the Attribute menu.  
Or, right-click and select Menubar from the popup menu.  
The Menubar / WindowPolicy Editor appears, as shown in Figure 4-1.



**Figure 4-1** Menubar / WindowPolicy Editor.

**To add a 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 ENTER 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 item to a menu:**

1. Click the Promote button from the Menu Editor dialog. The selected menu shifts to the right.
2. Type the menu item. It appears indented under its menu title.
3. Click the Demote button when you have entered all the items for that 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 ENTER to enter the menu item.
2. Click the Promote button, and type the first submenu item. Press ENTER.
3. Click the Demote 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 Insert 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 application 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 a tilde “~” in front of the letter you want to be the mnemonic key. Under Windows, you may also use the ampersand “&” key.

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, and type the desired letter.
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 Demote or Promote button.
- To move an item to another place in the menu, select the item and click the Shift Up or Shift Down 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 Insert button. An empty space will appear. Enter the new item, and press ENTER.
- To delete a menu item, select the line that you want to delete, and click the Delete button.
- To remove a menu bar from a window, remove all the items from the menu bar.

**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. This selector should be a unary message selector (no colons). WindowBuilder Pro will automatically generate matching skeletal methods for you.

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

**To test a menu:**

- Choose Test Menu. A new window will appear, with a working example of your menu in it.

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.

## Creating WindowPolicies

WindowPolicies provide a convenient mechanism for sharing menu definitions between multiple windows. WindowPolicy definitions may split a menubar definition across three categories - Standard Left, Standard Right and Application. Standard Left menus are commonly the File and Edit menus. Standard Right menus may include Window and Help menus. Application menus are specific to the application. This categorization allows you to set up inheritance hierarchies among the WindowPolicies. An abstract superclass may define the Standard Left and Right menus while subclasses define Application specific menus.

After launching the Menubar Editor, WindowPolicy use may be specified by checking the Use Window Policy check box. This will enable an number of additional fields in the editor. The Window Policy group to the upper left allows you to select a WindowPolicy class to be assigned to the window. WindowPolicies whose names are bracketed - <Name> - are system WindowPolicies. They may be assigned and viewed, but not edited. WindowPolicies may be created, saved and browsed (using the standard code browser) via the New, Save and Browse buttons.

The upper left combobox allows you to change the current group/category. The Set Group button allows you to move existing menus between groups. This makes it easy to

take an existing window with a menu definition and re-save the menu definition as a WindowPolicy.

WindowPolicies may be created in the context of designing a specific window, or they may be created as standalone classes independent of any specific window. When launched using the Menus command from the Attribute menu or via the Menu bar button on the Attribute panel, the editor look as it does in Figure 4-1. When launched via the WindowPolicies command in the File menu or in the Transcript menu, the editor looks appears as is shown in Figure 4-2.

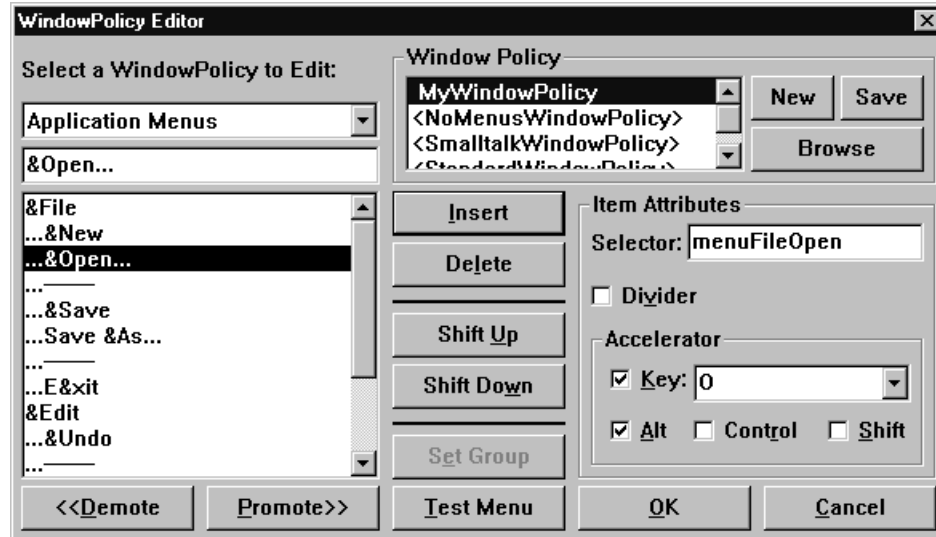


Figure 4-2 WindowPolicy Editor.

## Popup Menus on Widgets

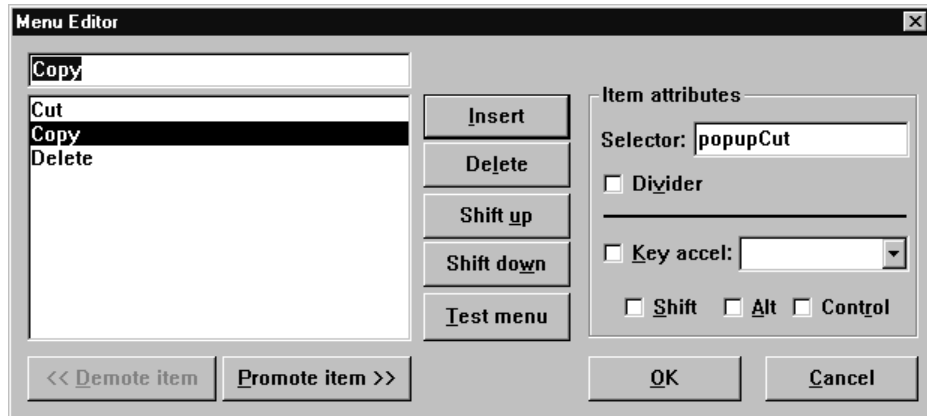
Most widgets can have popup menus associated with them. The process of creating a popup menu for a widget is the same as creating a menubar for a window. There are two minor differences:

- 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 to which you want to add a menu.
2. Click the Menu button on the Attribute panel.  
Or, select the Menus command from the Attribute menu.  
Or, right-click and select Menu from the popup menu.

The Menu Editor appears, as shown in Figure 4-3.



**Figure 4-3** Popup Menu Editor.

The process of creating a popup menu is the same as that of creating a menubar. For more information, refer to the sections on creating and editing menu bars.

## Chapter 5 Coding in WindowBuilder Pro

In this chapter, 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 in Figure 5-1 takes place. WindowBuilder Pro generated methods are shown in gray.

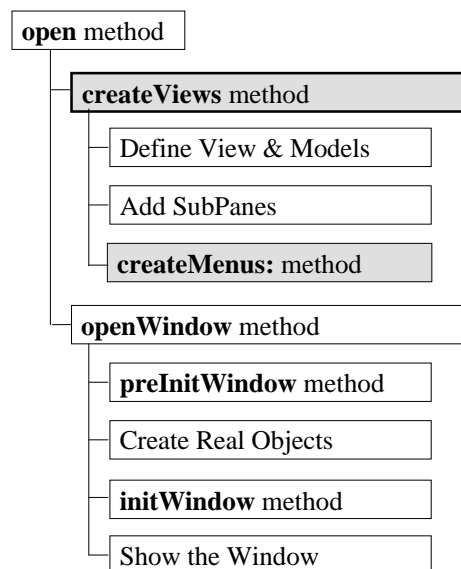


Figure 5-1 Steps in Processing and Opening a Window.



Essentially, the `#createViews` method generated by WindowBuilder Pro gets called by the `#open` method for the class you are constructing. This `#open` method, in essence, calls `#createViews` followed by `#openWindow`. If a window defines a menubar, the `#createViews` method will call `#createMenus:`. The `#createMenus:` method is generated by WindowBuilder Pro and contains all of the menu definitions for the window. When using WindowPolicies, WindowBuilder Pro will generate a `#windowPolicyClass` method rather than a `#createMenus:` method.

Notice the comment at the beginning of the `#createViews` method. It includes a warning telling you 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 `#createViews` method, overwriting the existing one, if any. Later in this chapter, we'll see how to get around this necessary limitation.

As you can see in Figure 5-1, 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 VisualSmalltalk application. The order in which these methods is called is important.

The `#preInitWindow` method is not used very often in VisualSmalltalk applications. 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 physically created (Note that by the time `#initWindow` gets called, the window is already created, so it's too late to add other controls or elements to it).

The `#initWindow` method, on the other hand, is one you may use extensively. This is an ideal method to use for such tasks as setting the contents of a pane with dynamically derived data that can't be hardcoded because it isn't known until the program executes.

The `#preInitWindow` and `#initWindow` methods enable you to do anything you want to a window or dialog without tampering with the `#createViews` method generated by WindowBuilder Pro.

## 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 `staticText` and `entryField`. 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 `#createViews` 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 paneNamed: 'promptText') contents: promptString.
  (self paneNamed: 'editor') contents: 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 now 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` 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
  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: 'Doooooooooh!') 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` method as follows:

```
ok
    responseText := (self paneNamed: 'editor') contents.
    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
    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 that you wish to interact with programatically must have a name. By default, WindowBuilder Pro does not provide widgets with names. In general, you will want to

give your widgets 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 #paneNamed: protocol like this:

```
(self paneNamed: 'nameField')
```

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

```
(self paneNamed: 'nameField') contents: aNameString
```

WindowBuilder Pro 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). If you do this, the above expression can be re-written as:

```
nameField contents: 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 ListBox and a StaticText, and that we want the selected item of the list to update the contents of the label. This code can be expressed as:

```
(self paneNamed: 'aStaticText') contents:
  (self paneNamed: 'aListBox') selectedItem
```

or more directly using instance variable assigned widgets as:

```
aStaticText contents: aListBox selectedItem
```

## Alternate window opening protocols

In addition to the standard #open protocol for launching windows, WindowBuilder Pro provides a number of other ViewManager protocols that can be used for opening windows (or modifying their opening characteristics).

### **centeredOnMouse**

Open the receiver centered over the mouse. Call this method from within #preInitWindow.

### **centeredOnScreen**

Open the receiver centered on the screen. Call this method from within #preInitWindow.

### **open**

Open the receiver.

**openAsMDIParent**

Open the receiver as an MDI parent window. MDI support must be installed.

**openCenteredOnMouse**

Open the receiver centered over the mouse.

**openRelativeTo:** *aViewManager* **offset:** *aPoint*

Open the receiver relative to *aViewManager*.

**openWithMDIParent:** *aViewManager*

Open the receiver with *aViewManager* as an MDI Parent. MDI support must be installed.

**openWithMyParent:** *aViewManager*

Open the receiver as a child of *aViewManager*'s parent (a sibling).

**openWithParent:** *aViewManager*

Open the receiver with *aViewManager* as Parent.

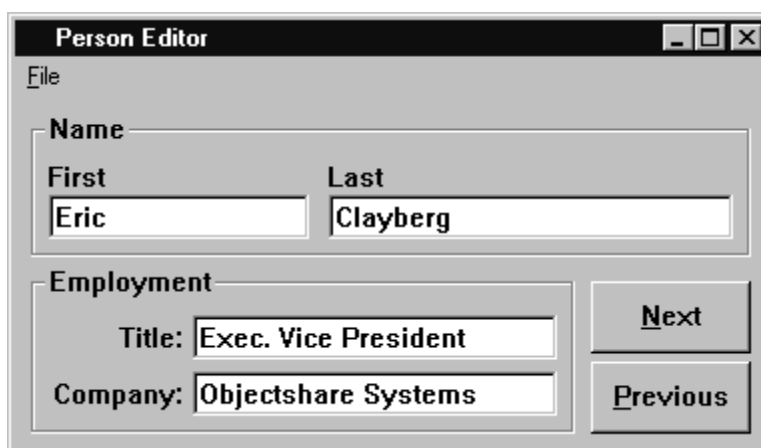
**positionRelativeTo:** *aViewManager* **offset:** *aPoint*

Open the receiver relative to *aViewManager*. Call this method from within `#preInitWindow`.

## Chapter 6 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 itself, were built using the same techniques. If you are interested in further, more sophisticated examples, examine the windows that make up WindowBuilder Pro itself.

The finished Person Editor application is shown in Figure 6-1.



**Figure 6-1. 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. To be consistent with the Overview chapter, the steps are divided into two sections: Designing the Interface, and Attaching Callbacks to Widgets. You probably would not actually create an application in this manner; you would more likely mix designing the interface, attaching callbacks, and event handlers as you think of them.

Positions and sizes are given so that you may exactly duplicate the application in Figure 6-1. In order to achieve this level of precision, it will probably be necessary to use the Size and Position buttons. If exactly duplicating the screen is not important to you, you can just place the widgets by eye, using the mouse. A working version of the Person Editor application (in class `PersonEditor`) is provided with WindowBuilder Pro. So, if you want, you can just display the Person Editor in WindowBuilder Pro and follow along with the text.

Be sure to save the application occasionally as you work. The first time you do, you are prompted to save your window as a subclass of `ViewManager` (or `ApplicationCoordinator`).

## Designing the Interface

1. From the File menu, choose New.
2. Resize the window to 384 x 220 by clicking the Size button on the Attribute panel. Type **Person Editor** in the title field.
3. Right click anywhere in the layout pane. Bring up the Menubar Editor by choosing Menubar from the popup menu. Build the menu shown in Figure 6-2. Recall that mnemonic keys are displayed as underlined characters in menu items, and accelerator keys are displayed on the right of the menu. Using Figure 6-2 as a guide, add accelerator and mnemonic keys to the menu.



**Figure 6-2 Person Editor Application Menu.**

4. Place an EnhancedEntryField widget at position 16,48 with a size of 132x24. Name the widget `firstName` by typing **firstName** in the name field on the attribute panel.
5. Place an EnhancedEntryField widget at position 156,48, with a size of 204x24. Name the widget **lastName**.
6. Place a StaticText widget at position 16,32. Name the widget **firstNameLabel**. In the Text field, type **First**.
7. Place a StaticText widget at position 156,32. Name the widget **lastNameLabel**. Type **Last** in the Text field.

8. Place the first of two EnhancedEntryField 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 EnhancedEntryField widget below the first, using the left mouse button (which unloads the cursor). Select the top widget, and set its position to 88,108, and its size to 184x24.
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. Click a blank part of the window to deselect the widgets. Position the lower widget at 88,136.
10. Select the upper EnhancedEntryField widget, and name it **title**. Select the lower EnhancedEntryField widget, and name it **company**.
11. Place a StaticText widget at position 16,112. Name the widget **titleLabel**. Type **Title:** in the Text field. In the Style box, select rightJustified. Keep the widget selected.
12. Copy the titleLabel widget by choosing Copy from the Edit menu. Paste the copy at position 16,140. Name the new widget **companyLabel**. Type **Company:** in the Text field. Note that the Style setting is already correct, since you copied it from the titleLabel widget.
13. Select a GroupBox widget, and drag it so that it surrounds the firstName and lastName widgets and their labels. The position of the groupbox is 8,8, and its size is 360x72. Name the groupbox **nameBox**. Type **Name** in the Text field.
14. Select a GroupBox widget, and drag it so that it surrounds the title and company widgets and their labels. The position of the groupbox is 8,84, and its size is 272x84. Name the groupbox **employmentBox**. Type **Employment** in the Text field.
15. Place a PushButton at position 288,92, and set the size to 80x36. Name the button **next**. Place the cursor in the Text field, type &**Next**.
16. Place a PushButton at position 288,132. Name the button **previous**. Select the “next” button, then SHIFT-select the “previous” button. From the Size menu, choose Replicate Size, then Replicate Width. The buttons should now be the same size. Click elsewhere in the window to deselect the widgets. Select the “previous” button. In the Text field, type &**Previous**.
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 Edit window. 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 to the widgets to give the application some behavior.



## Attaching Callbacks

Right click anywhere in the layout pane, other than a widget. Bring up the Menubar Editor by choosing Menubar from the popup menu. For each menu item in the chart, type the corresponding method in the Selector box. When you are done, close the menubar Editor.

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

Select the firstName EnhancedEntryField widget. In the Send: field, type **textChanged:**. This attaches the textChanged callback to the firstName widget.

Using the chart below, attach callbacks for the remaining widgets in the same way as Step 2. The “firstName” callback is included for reference.

Widget Name	When:	Send:
firstName	textChanged	textChanged:
lastName	textChanged	textChanged:
title	textChanged	textChanged:
company	textChanged	textChanged:
next	clicked	next:
previous	clicked	previous:

Save the application. WindowBuilder Pro will regenerate the window’s layout and menu definitions (e.g., the #createViews and #createMenus: 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 PersonEditor class. You can either type them into your class definition or copy the ones that have been provided. Be sure to include all the methods provided in the PersonEditor class; they are all necessary to the proper operation of the application. 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 framing blocks (resize behavior) for each of the widgets as well as an overall tab order.

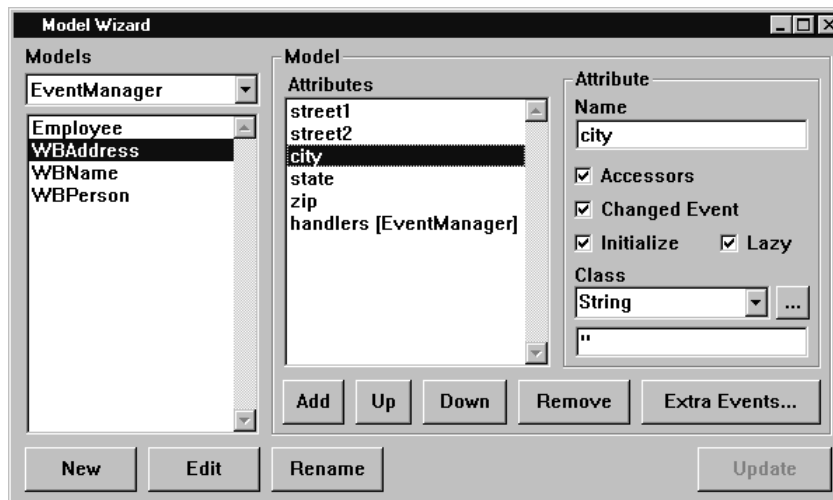
## Chapter 7 Model Objects

In addition to its powerful GUI building functions, WindowBuilder Pro also provides a number of facilities for building and managing domain model objects. Non-visual, event-driven objects may be added to WindowBuilder Pro built windows as easily as visual objects. This means that it is easy to utilize SharedValues, Timers and any other event driven objects like a Person object. New model objects (generally subclasses of EventManager) can be generated complete with attribute accessors and event tables. Once you have created a new model object, WindowBuilder Pro's Layout Wizard makes it easy to create a default screen layout for it that maps the model object's attributes to appropriate widgets and vice versa. This chapter will present each of these feature.

### The Model Wizard

The Model Wizard shown in Figure 7-1 allows you to create new subclasses of EventManager (or add attributes to subclasses of ViewManager, ApplicationCoordinator, or CompositePane).

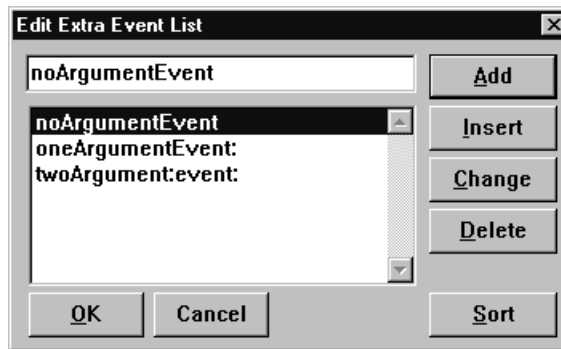
Model classes can be selected from the left-hand pane or created via the Add button. For each model class, attributes may be added, removed and ordered. Each attribute specifies a name; whether accessor methods are generated (standard get and set methods); whether a changed event is associated with the attribute (of the form #<name>Changed:); whether the attribute should be initialized (via an #initialize method or via lazy initialization in the get method); and what class type should be used to initialize the object (for some types, an object - string, symbol, boolean, etc. - may be specified).



**Figure 7-1 The Model Wizard.**

Model classes may be renamed via the Rename button and edited by hand in a standard browser via the Edit button.

The Model Wizard also lets you specify additional events above and beyond the changed events specified for each attribute by clicking on the Extra Events button. The Extra Event List editor shown in Figure 7-2. New events may be added, removed and renamed.

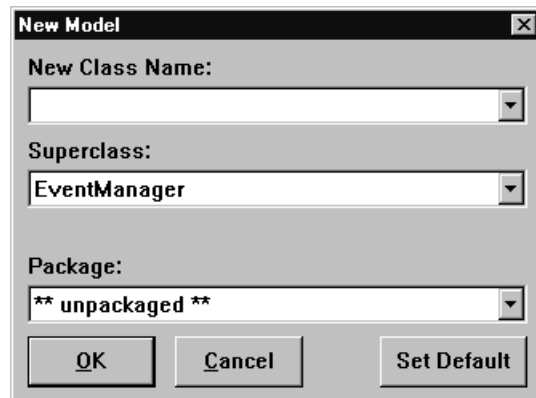


**Figure 7-2 Extra Event List Editor.**

To create a new model object:

1. Select the EventManager category from the combobox in the upper left corner of the Model Wizard.
2. Click the New button. This opens the New Model dialog shown in Figure 7-3.

3. Type in a name for the new Model class and click OK (you may also specify a superclass other than EventManager as well as a package/application in which to store the class definition if you are using Team/V, ENVY/Developer or XoteryX).



**Figure 7-2 New Model Dialog.**

Once you have created the model object class, you will want to add attributes to it.

**To create define a new attribute:**

1. Select the model object class in the listbox on the right.
2. Click the Add button to add a new attribute.
3. Enter a name for the attribute in the Name field.
4. If you would like the attribute to have standard accessors, check the Accessors checkbox. This will result in a get, set and basic set method being generated.
5. If you would like the attribute to trigger a changed event when it is modified, check the Changed Event checkbox. This will add a new event to the class's event table and create a method that triggers it. The standard set method will call this event triggering method.
6. If you would like the attribute to be initialized, check the Initialize checkbox. This will cause the attribute to be initialized in an `#initialize` method. If you would prefer that the attribute be initialized via lazy initialization in the get method, check the Lazy checkbox as well.
7. If you have chosen to have the attribute initialized, you may specify a default class type to initialize the attribute with. A number of common classes are provided in the Class combobox as well as any model classes you have defined. If you wish to specify a class that is not shown in the combobox, click on the '...' button and select an appropriate class from the list. Some simple class types like String,

Integer, Boolean, etc. allow you to specify an actual value with which to initialize the attribute. This value may be entered in the entryfield below the Class combobox.

When you have finished defining attributes, click on the Update button to save your work and generate all of the specified code.

To illustrate what kind of code WindowBuilder Pro will generate for you, the WBName sample class is shown below:

```
WLObject subclass: #WBName
  instanceVariableNames:
    ' first middle last '
  classVariableNames: ''
  poolDictionaries: '' !

!WBName class methods !
constructEventsTriggered
  "Private - answer all the events which can be triggered
   by instances of the receiver.
  Remove the #generated tag before modifying this method.
  (Generated by WindowBuilder Pro)"

  #generated.
  ^super constructEventsTriggered
    add: #firstChanged;;
    add: #middleChanged;;
    add: #lastChanged;;
    yourself!

wbBasicVersion
  "Private - Answer the WindowBuilder Pro version.
   Generated in: VisualSmalltalk Enterprise 3.1.0"

  ^3.1! !

!WBName methods !
basicFirst: aString
  "Private - set the value of first to <aString>.
  Remove the #generated tag before modifying this method.
  (Generated by WindowBuilder Pro)"

  #generated.
  first := aString!
```

```
basicLast: aString
    "Private - set the value of last to <aString>.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    last := aString!

basicMiddle: aString
    "Private - set the value of middle to <aString>.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    middle := aString!

first
    "Answer the value of first.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    ^first ifNil: [first := '']!

first: aString
    "Set the value of first to <aString>.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    first = aString
        ifFalse: [
            self
                basicFirst: aString;
                changed;
                triggerFirstChangedEvent]!

last
    "Answer the value of last.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    ^last ifNil: [last := '']!
```

```
last: aString
    "Set the value of last to <aString>.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    last = aString
        ifFalse: [
            self
                basicLast: aString;
                changed;
                triggerLastChangedEvent]!

middle
    "Answer the value of middle.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    ^middle ifNil: [middle := '']!

middle: aString
    "Set the value of middle to <aString>.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    middle = aString
        ifFalse: [
            self
                basicMiddle: aString;
                changed;
                triggerMiddleChangedEvent]!

triggerFirstChangedEvent
    "Private - trigger the #firstChanged: event.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    self triggerEvent: #firstChanged: with: self first!
```

```

triggerLastChangedEvent
    "Private - trigger the #lastChanged: event.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    self triggerEvent: #lastChanged: with: self last!

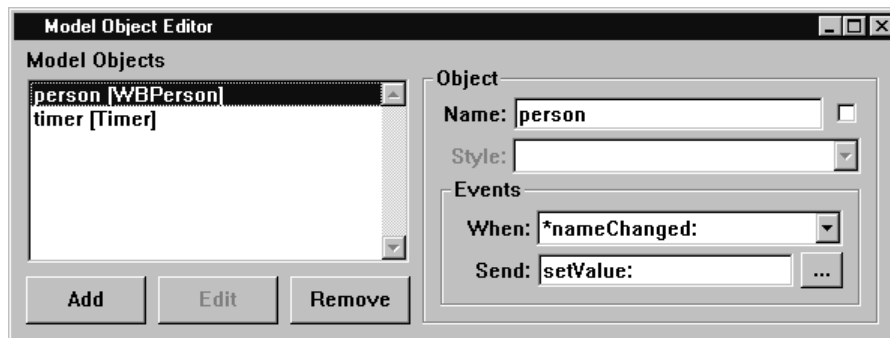
triggerMiddleChangedEvent
    "Private - trigger the #middleChanged: event.
    Remove the #generated tag before modifying this method.
    (Generated by WindowBuilder Pro)"

    #generated.
    self triggerEvent: #middleChanged: with: self middle! !

```

## The Model Object Editor

The Model Object Editor shown in Figure 7-3 allows you to attach non-visual, event-driven objects (e.g., EventManager subclasses) to a window.



**Figure 7-3 Model Object Editor.**

Model objects may be added, removed and edited (if there is an editor associated with the class). Model objects may be given names, assigned to instance variables, or given callbacks.

**To attach a model object to a window:**

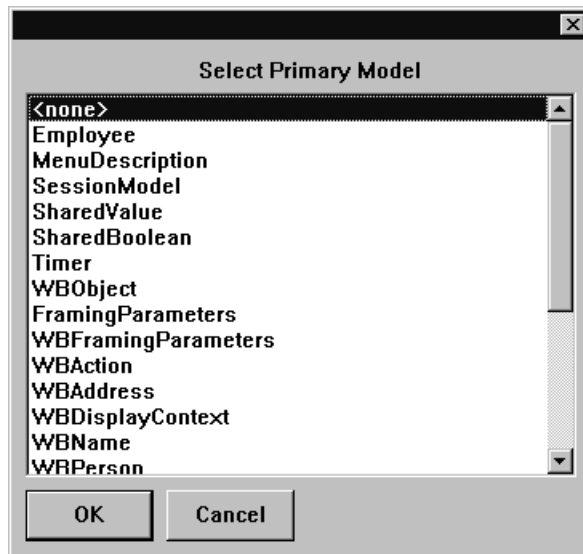
1. Click the Add button and select an appropriate model class from the list. This will create a new unnamed instance of that model class.
2. Give the instance a name



3. The object may be specified as an instance variable of the window by clicking on the checkbox to the right of the Name: entryfield.
4. Select an appropriate style for the object (if the object supports styles)
5. Specify event handlers for the object's events. Simple event handler that hook the object to the window may be specified via the When: and Send: fields (just as in the main WindowBuilder Pro editor). For more complex events (or to attach multiple handlers to each event), click on the '...' button to launch the Event Manager.

## Primary Models

Many windows that you will build will have a primary domain model object associated with them. WindowBuilder Pro makes it easy to establish this association and generate code to facilitate its use. The Select Primary Model Dialog is shown in Figure 7-4.



**Figure 7-4** Select Primary Model Dialog.

The default selection - '<none>' - means that there is no primary model object associated with the window.

Selecting a primary model object will cause WindowBuilder Pro to add an instance variable called 'model' to the window's class definition and assign an instance of the primary model object class to it. An `#initialize` method will be generated that

initializes the model instance variable to an instance of the primary model class. An `#openOn:` method is also generated to make it easy to open the window on a specific instance of the model object class.

Once the primary model has been established, it may be attached to any object in the window using the Model Object Editor or the Event Manager.

## The Layout Wizard

Once you have developed a domain model object you will likely want to build a screen to represent it. The Layout Wizard shown in Figure 7-5 can generate default layouts based on any domain model class (e.g., `EventManager` subclasses) or the result of any Smalltalk expression.

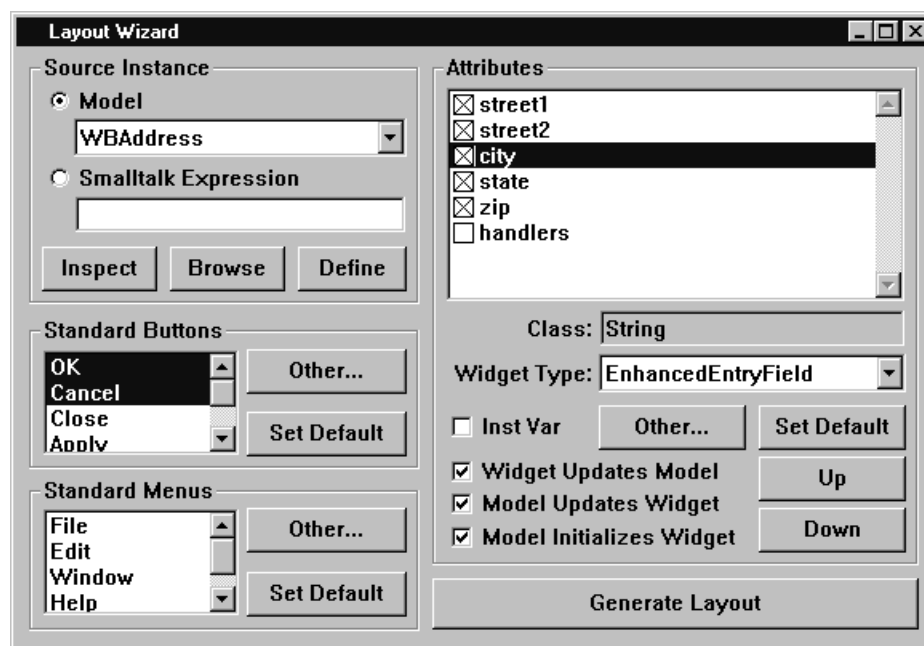


Figure 7-5 The Layout Wizard.

The source instance may be specified by selecting an existing Model class or by entering an arbitrary Smalltalk expression that evaluates to an appropriate instance. The source instance may be inspected, its class opened in a browser or its definition modified via the Model Wizard.

Once the source instance has been specified, its attributes are listed on the right-hand side of the window. Attributes may be included or not by checking on or off the checkboxes next to their names. Default widget types are provided for most data types (e.g., `CheckBox` for `Boolean`, `EnhancedEntryField` for `String`, etc.). Widget types may be easily changed to customize the layout. If the source instance is an instance of a well defined model class (with getters, setters and events), the bi-directional links may be established between the model and the widgets.

Standard buttons (e.g., `OK`, `Cancel`, etc.) may be added automatically as well as a default menubar with pre-defined `File` and `Edit` menus. The `Generate Layout` button will create a widget layout in the current window, dialog or `compositepane` based on the attributes specified.

**To generate a default layout:**

1. Create a new window, dialog or `compositepane` to hold the generated widgets.
2. Launch the `Layout Wizard` and specify an appropriate source instance by selecting a model class or entering a `Smalltalk` expression.
3. Select the attributes in the right hand list for which widgets should be generated by checking the checkboxes next to their names. The `Layout Wizard` will establish an initial state for each attribute based on evaluating the instance and its class.
4. For each attribute, specify a widget type. The `Layout Wizard` will have specified defaults for each attribute based on its data type. Widget types appropriate for a given data type are listed in the `Widget Type: combobox`. If a different widget type is required, click the `Other` button to select an arbitrary widget type. If you would like to establish a new default widget type for a given data type, click on the `Set Default` button.
5. If the widget should be added as an instance variable of the window, check the `Inst Var` checkbox.
6. If the model object defines a `set` method for the attribute and if you would like the widget to automatically update the model when its value changes, check the `Widget Updates Model` checkbox. Note that checking on any of the `Widget/Model` options will cause `WindowBuilder Pro` to associate the source instance to the window as the primary model.
7. If the model object defines a `changed` event for the attribute (of the form `#<name>Changed:`) and if you would like the model to automatically update the widget when its attribute changes, check the `Model Updates Widget` checkbox. This also assumes that the widget responds appropriately to the `#setValue:` message.

8. If you would like the model to initialize the contents of the widget, then check the Model Initializes Widget checkbox. This requires the Widget Updates Model checkbox to be checked as well.
9. The widgets are created in the same order as the attributes. Re-arrange the attributes by selecting an attribute and clicking the Up or Down buttons
10. If the window should have standard buttons created for it across the bottom of the window, select the button names in the Standard Buttons listbox (OK & Cancel are selected by default).
11. If you would like to add buttons that aren't listed, click on the Other button to specify additional ones. Clicking the Set Default button will establish the default set of standard buttons that will be used in subsequent uses of the Layout Wizard.
12. If the window should have a default menu bar (only permitted for windows), select the menu names in the Standard Menus listbox. Many of the menus listed are provided with default templates that will be generated as well (e.g., the File menu will automatically define New, Open, Save, Save As and Exit items).
13. If you would like to add menus that aren't listed, click on the Other button to specify additional ones. Clicking the Set Default button will establish the default set of standard menus that will be used in subsequent uses of the Layout Wizard.
14. Once all of the options have been specified, click the Generate Layout button to create all of the widgets and menus into the currently edited window
15. Edit and save the window using the standard WindowBuilder Pro techniques.

The result of using the Layout Wizard against the WBAAddress sample class is shown in Figure 7-6.









































WLObject subclass: #WBAAddress  
 instanceVariableNames:  
   'street1 street2 city state zip'  
 classVariableNames: "  
 poolDictionaries: "



**Figure 7-6** Layout Wizard results.



## Chapter 8 Toolbar Reference

	Test Window		Set Font
	New Window		Set Color
	New Dialog		Set Framing
	Open Window		Set Menu
	Save		PARTS Interface
	Cut		Set Attributes
	Copy		Edit Code
	Paste		Tabbing/Group Editor
	Delete		Text Items
	Send To Back		Buttons
	Bring To Front		Lists
	Duplicate		Valuators
	Autosize		Grouping Items
	Replicate Width		Miscellaneous Items
	Replicate Height		CompositePanels
	Align Left		Windows 95
	Align Horizontal Center		
	Align Right		
	Align Top		
	Align Vertical Center		
	Align Bottom		
	Distribute Vertically		
	Distribute Horizontally		
	Show Grid/Set Grid Size		

	StaticText		WBStaticGraphic
	EntryField		GraphPane
	EnhancedEntryField		AnimationPane
	TextEdit		WBToolBar
	TextPane		StatusPane
	Button		VideoPane
	DrawnButton		RadioButtonGroup
	RadioButton		CheckBoxGroup
	CheckBox		EntryFieldGroup
	ThreeStateButton		SexPane
	ListBox		Header
	ListPane		ListView
	MultipleSelectListBox		ProgressBar
	ComboBox		RichEdit
	ButtonListBox		StatusWindow
	ColumnarListBox		TabControl
	SpinButton		TrackBar
	ScrollBar (horizontal)		TreeView
	ScrollBar (vertical)		UpDown
	GroupBox		
	StaticBox		

---

# Chapter 9 Menu Reference

## Transcript

### **New Window**

Create a new window. This causes WindowBuilder Pro to from scratch on a new copy of the default template window.

### **New Dialog**

Create a new dialog. This causes WindowBuilder Pro to from scratch on a new copy of the default template dialog

### **New CompositePane**

Create a new dialog. This causes WindowBuilder Pro to from scratch on a new copy of the default template compositepane.

### **Edit Window...**

Edit an existing ViewManager, WindowDialog, ApplicationCoordinator or CompositePane. WindowBuilder Pro will prompt the user with a class dialog containing a list of viewmanagers, applicationcoordinators and compositepanes that have been created by it.

### **Models...**

Create and edit domain model objects. The Model Wizard allows you to create new subclasses of EventManager (or add attributes to subclasses of ViewManager, ApplicationCoordinator, or CompositePane).

### **WindowPolicies...**

Create and edit domain WindowPolicy subclasses. WindowPolicies provide a convenient mechanism for sharing menu defintions between multiple windows. WindowPolicy definitions may split a menubar definition across three categories - Standard Left, Standard Right and Application.



**Templates...**

Opens the Template Editor. The Template editor allows you to specify default attribute values for attributes of all the widgets supported by WindowBuilder Pro.

**Properties...**

Displays the Property Editor. The property 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.

**Manager**

Displays a submenu with the following choices:

**Add-In Manager**

Launch the Add-In Manager dialog. The Add-In Manager is used to extend the WindowBuilder Pro environment with additional functionality.

**Bitmap Manager**

Launch the Bitmap Manager window. The Bitmap Manager is used to manage bitmaps and bitmap pool dictionaries. See the *Pool Managers* chapter for complete details.

**Font Manager**

Launch the Font Manager window. The Font Manager is used to manage fonts and font pool dictionaries. See the *Pool Managers* chapter for complete details.

**NLS Manager**

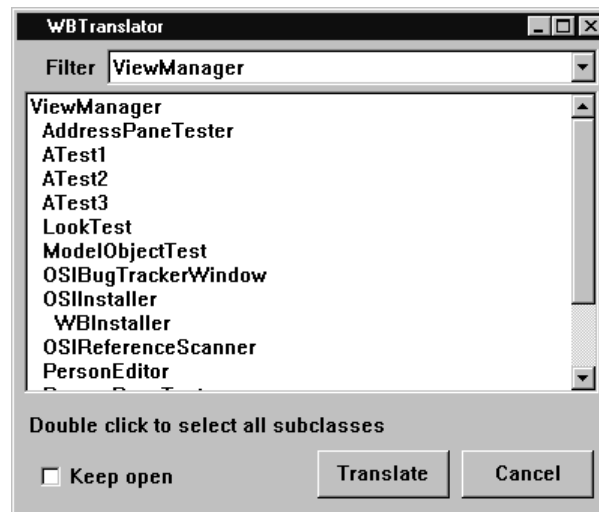
Launch the NLS Manager window. The NLS Manager is used to manage NLS string dictionaries and language mappings. See the *Pool Managers* chapter for complete details.

**Translator**

Displays a submenu with the following choices:

**Standard Translation...**

Opens the standard translation utility. This utility facilitates translating `#createViews` methods from windows built with older versions of WindowBuilder Pro to the code generation style in use in version 3.1.

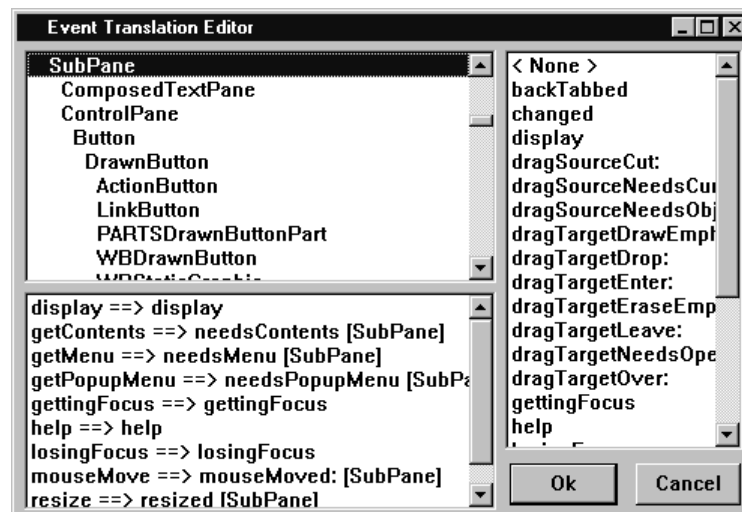


### Event Translation...

Opens the event translation utility. This utility facilitates translating windows built using the old event model (from Smalltalk/V 2.0) to using the new event model in VisualSmalltalk.

### Event Mappings...

Opens the event mapping utility. This utility is used to set up the old to new event maps used by the Event Translation utility.



When editing the event maps, class names appear in the upper left list box, old to new event mappings appear in the lower left list box, and new events appear in the right hand list box. Select the desired class in the upper left list box and the old to new event mappings should appear in the lower left list box. Old events mapped to new events appear as

oldEventName ==> newEventName

Old events that are mapped to new events in a class are inherited by all subclasses. Thus if you have “Button” selected in the upper left list box, you would see

display ==> display [SubPane]

in the lower left list box, indicating that the old to new event mapping of #display to #display is inherited from SubPane.

Old events that are not mapped appear as

oldEventName

You can change the old to new event mapping by selecting one of the old events in the lower left list box, then selecting the desired new event in the right list box.

### **About...**

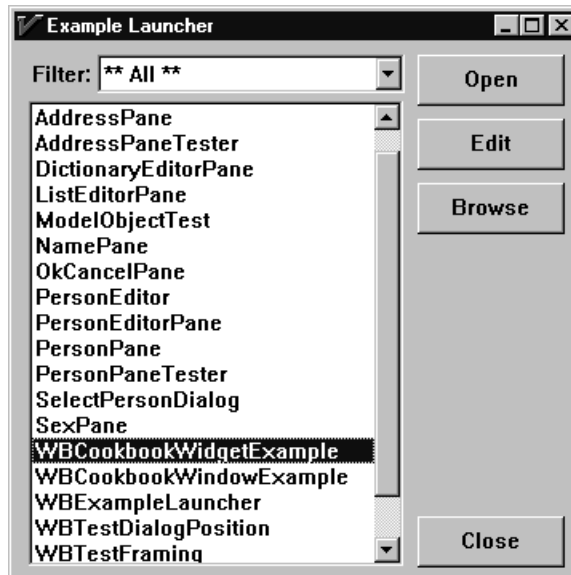
Displays information about WindowBuilder Pro.

### **Installed Products...**

Displays information about installed products.

## Examples...

Opens the Example Launcher window. This gives you a fast path to opening, editing and browsing any of the examples provided with WindowBuilder Pro or any add-on widget products. The various examples are grouped into multiple categories with individual examples falling into one or more category.



# File

## New Window

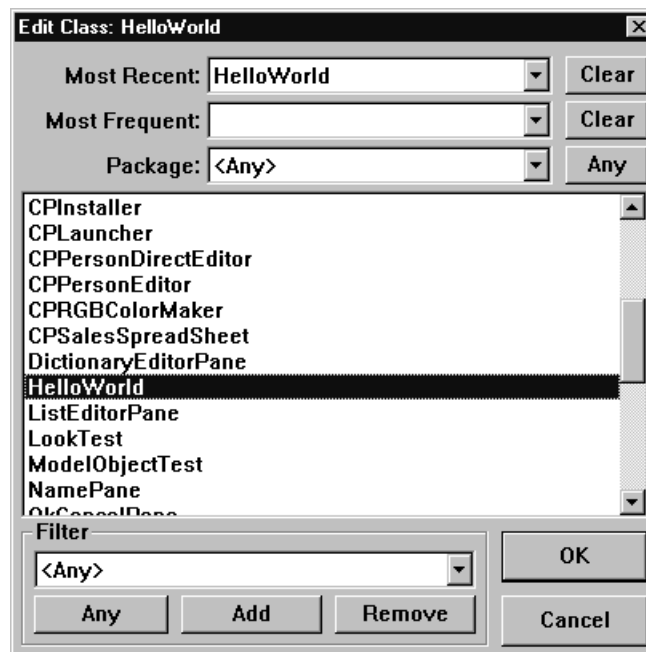
Create a new window. This causes WindowBuilder Pro to discard any work in progress, and start from scratch on a new copy of the default template window.

## New Dialog

Create a new dialog. This causes WindowBuilder Pro to discard any work in progress, and start from scratch on a new copy of the default template dialog.

## Open...

Edit an existing ViewManager, WindowDialog, ApplicationCoordinator or CompositePane. WindowBuilder Pro will prompt the user with a class dialog containing a list of viewmanagers, applicationcoordinators and compositepanes that have been created by it (using this class dialog, you may also attempt to edit other windows, as long as they have a `#createViews` or `#open` method). When a window is chosen, WindowBuilder Pro will discard any work in progress, and begin editing the layout of the selected window.



By default, the list in this dialog contains only those windows built by WindowBuilder Pro. The classes are listed alphabetically. Handy package/application and class type filters are provided to help you look at just the classes that you wish. The package/application filter is only available if you are using Team/V, ENVY/Developer or XoteryX. As a convenience, WindowBuilder Pro keeps track of the most recently accessed and most frequently accessed classes.

## Composite Panes

Displays a submenu with the following choices:

### New

Create a new CompositePane. This causes WindowBuilder Pro to discard any work in progress, and start from scratch on a new copy of the default template compositepane.

### Open...

Edit an existing CompositePane. WindowBuilder Pro will prompt the user with a class dialog containing a list of compositepanes that have been created by it. When a compositepane is chosen, WindowBuilder Pro will discard any work in progress, and begin editing the window of the selected CompositePane.

### Create

Create a new CompositePane from the widgets selected in the editing window. This causes another copy of WindowBuilder Pro to open with the selected widgets as the basis for a new CompositePane. If this compositepane is saved, you will have the opportunity to replace the existing widgets with the new compositepane.

### Ungroup

Split the selected compositepane into its component parts.

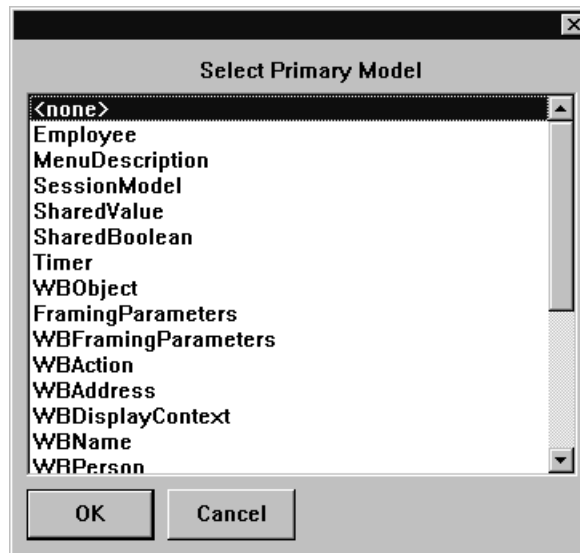
**Warning:** any behavior associated with the compositepane's class will be lost. The component widgets will preserve their names and the names of any event handlers they had.

## Model Objects

Displays a submenu with the following choices:

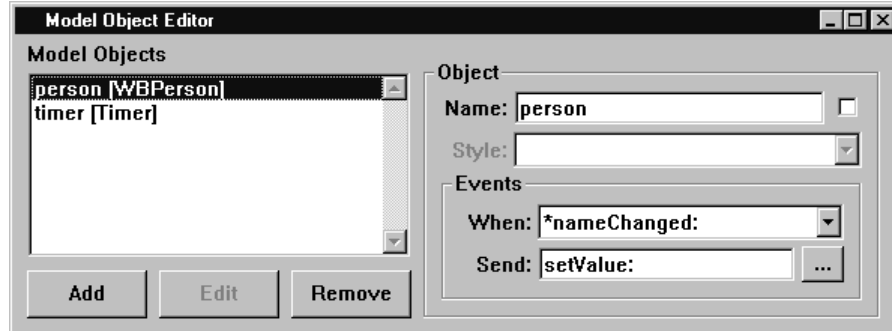
### Select Primary Model...

Assign a primary model class to the window. This will automatically add a “model” instance variable to the window’s class definition along with an #initialize and a #openOn: method.



### Edit Models...

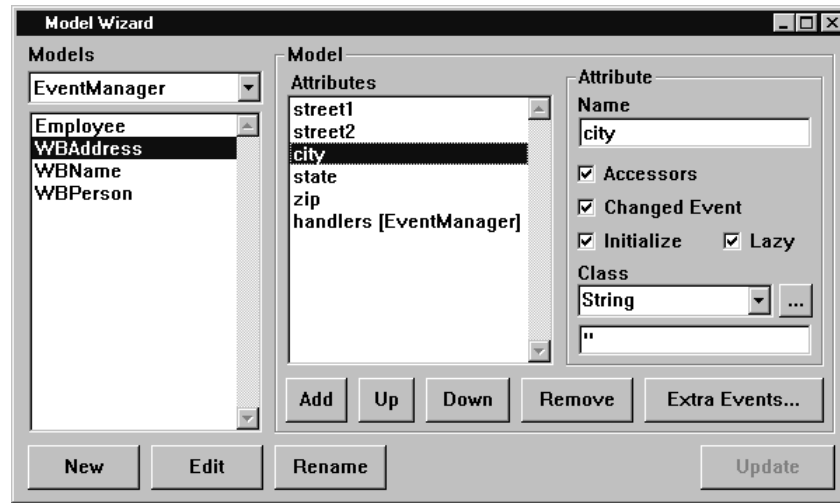
Edit the model objects associated with the window. Any non-visual event driven object may be managed by WindowBuilder Pro.



Model objects may be added, removed and edited (if there is an editor associated with the class). Model objects may be given names, assigned to instance variables, or given callbacks.

## Model Wizard...

Create and edit domain model objects. The Model Wizard allows you to create new subclasses of EventManager (or add attributes to subclasses of ViewManager, ApplicationCoordinator, or CompositePane). Model classes can be selected from the left-hand pane or created via the Add button. For each model class, attributes may be added, removed and ordered. Each attribute specifies a name; whether accessor methods are generated (standard get and set methods); whether a changed event is associated with the attribute (of the form #<name>Changed:); whether the attribute should be initialized (via an #initialize method or via lazy initialization in the get method); and what class type should be used to initialize the object (for some types, an object - string, symbol, boolean, etc. - may be specified).

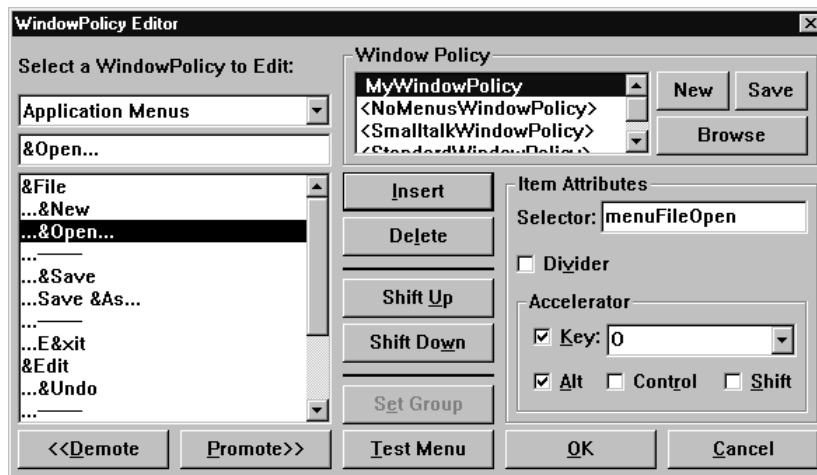


The Model Wizard also lets you specify additional events above and beyond the changed events specified for each attribute. Model classes may be renamed via the Rename button and edited by hand in a standard browser via the Edit button.

## WindowPolicies...

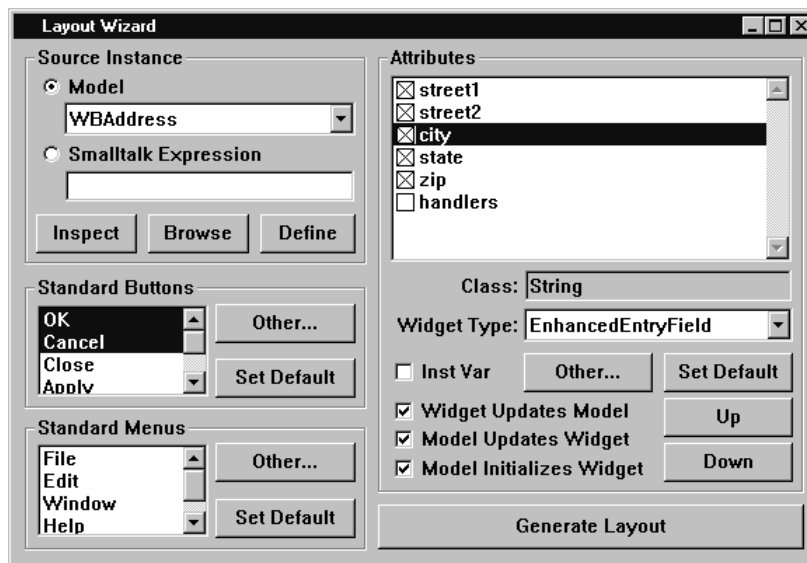
Create and edit domain WindowPolicy subclasses. WindowPolicies provide a convenient mechanism for sharing menu definitions between multiple windows. WindowPolicy definitions may split a menubar definition across three categories - Standard Left, Standard Right and Application. Standard Left menus are commonly the File and Edit menus. Standard Right menus may include Window and Help menus. Application menus are specific to the application.





### Layout Wizard...

Build default layouts based on arbitrary Smalltalk instances. The Layout Wizard can generate default layouts based on any domain model class (e.g., EventManager subclasses) or the result of any Smalltalk expression. The source instance may be inspected, its class opened in a browser or its definition modified via the Model Wizard.



Once the source instance has been specified, its attributes are listed on the right-hand side of the window. Attributes may be included or not by checking on or off the checkboxes next to their names. Default widget types are provided for most data types

(e.g., CheckBox for Boolean, EnhancedEntryField for String, etc.). Widget types may be easily changed to customize the layout. If the source instance is an instance of a well defined model class (with getters, setters and events), the bi-directional links may be established between the model and the widgets.

Standard buttons (e.g., OK, Cancel, etc.) may be added automatically as well as a default menubar with pre-defined File and Edit menus. The Generate Layout button will create a widget layout in the current window, dialog or compositepane based on the attributes specified.

## Save

Save the edited viewmanager, applicationcoordinator or compositepane by generating the appropriate code.

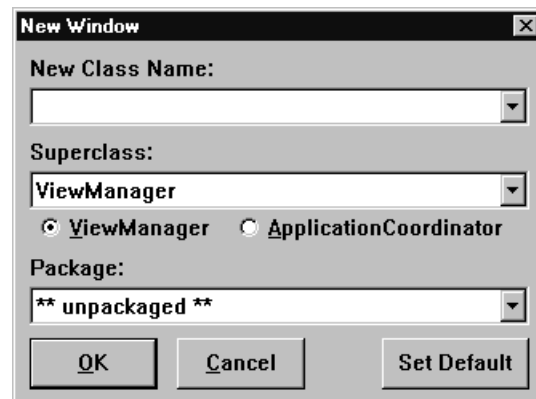
If the edited object has not been associated with a class, this command behaves exactly as the **Save As...** command, described below.

**Warning:** when a viewmanager, applicationcoordinator or compositepane's definition is saved, it writes over the previous definitions method for the window's layout. You will have to resort to the change log to recover code that is lost in this manner.

## Save As...

Save the edited viewmanager, applicationcoordinator or compositepane 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:



In the top combobox, enter a name for the new viewmanager (windowdialog, applicationcoordinator, compositepane) you're creating; if you wish to save over an existing class, the names of all the available viewmanagers (windowdialogs,

applicationcoordinators, compositepanes) can be chosen from this combobox. If you want a new class to have a different superclass than ViewManager (WindowDialog, ApplicationCoordinator, CompositePane), select it in the combobox below. If you are using Team/V, ENVY/Developer or XoteryX, you may also specify a package or application in which to save the class definition.

**Warning:** when a window's definition is saved, it writes over the previous method's definitions for the ViewManager's, ApplicationCoordinator's or CompositePane's layout. You will have to resort to the change log to recover code that might be lost in this manner.

## Save As Default

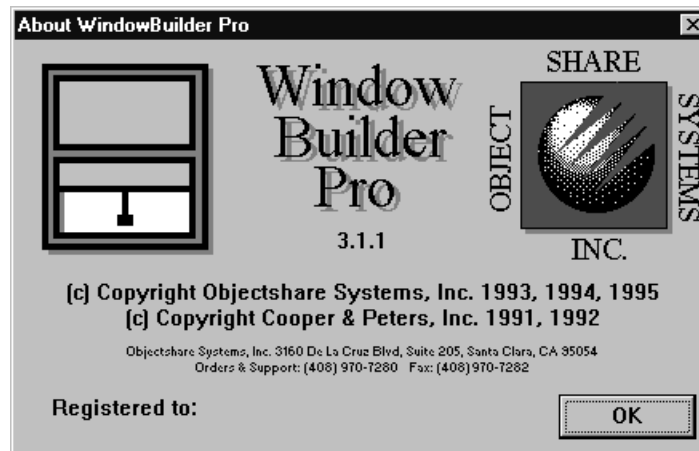
Save the current viewmanager (windowdialog, applicationcoordinator or compositepane) as the default template. Any subsequent windows that are created will use this template as the starting point.

## 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. Closing WindowBuilder Pro with the system closebox will have the same effect.

## About...

Displays information about WindowBuilder Pro.



## Installed Products...

Displays information about installed products.



# Edit

## Undo

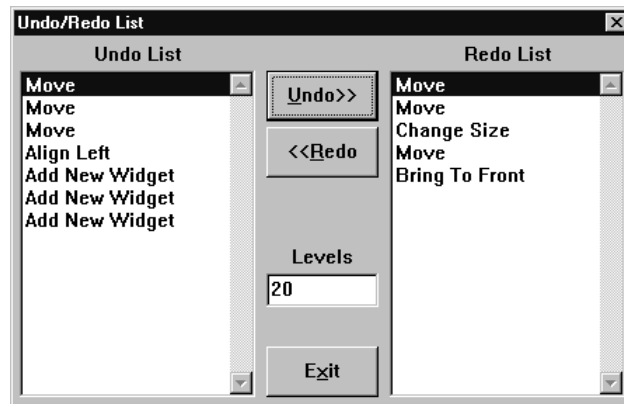
Undoes the last edit operation. This command is dimmed when an operation cannot be undone. The number of available undo levels is configurable from the WindowBuilder Pro Property editor.

## Redo

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



## Cut

Remove the selected widget(s) from the interface, placing it (them) on the clipboard for later pasting.

## Copy

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

## Paste

Load the cursor with the widgets on the clipboard. When the mouse is clicked somewhere within the layout pane, the clipboard selection will be placed at that location.

**Clear**

Remove the selected widgets(s) from the interface without affecting the clipboard.

**Select**

Displays a submenu with the following choices:

**All**

Selects all the widgets within the main window.

**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.

**All In Same Hierarchy**

Displays all the widgets in the same hierarchy.

**Bring To Front**

Move the selected widget in front of all the other widgets.

**Send To Back**

Move the selected widget behind all the other widgets.

**Bring Forward**

Moves the selected widget forward one position in the Z-order.

**Send Backward**

Moves the selected widget backward one position in the Z-order.

**Duplicate**

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

**Morph...**

Morph the selected widget into a different type while preserving as many attributes as possible. Attributes not needed by the target class will be lost. Attributes not provided

by the source class will be defaulted. The popup menu for each widget provides a list of ideal morphing candidates.

**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 `#needsContents` Callback for a `ListBox` would not be appropriate for an `EntryField`. It is recommended that morphing be limited to similar classes of objects.

### Browse Class...

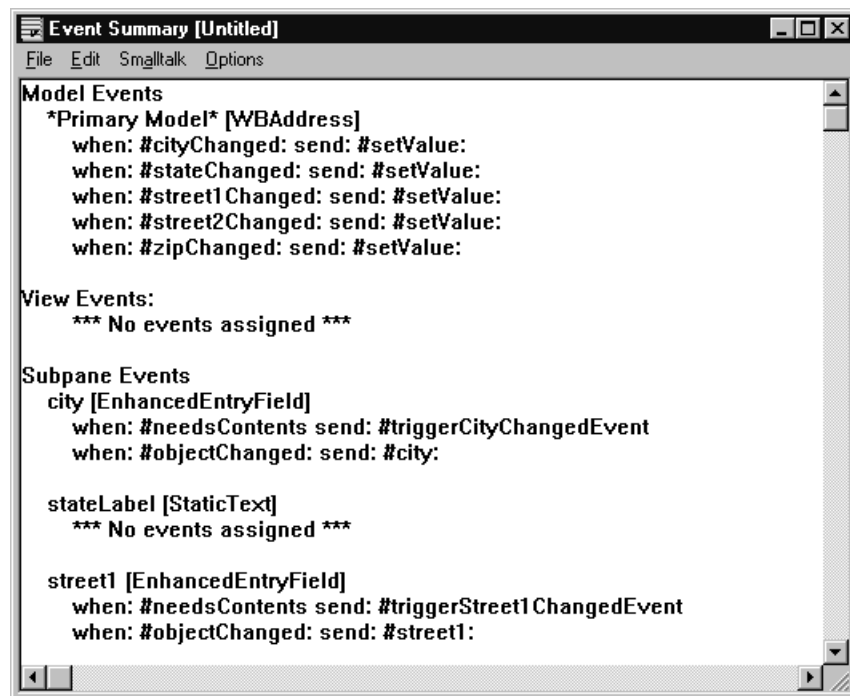
Display a standard class browser on the currently edited window. If the class has not been saved yet, you will be prompted to save the class.

### Browse Widget Class...

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

### Event Summary...

Launch a special text window containing a summary of the events used within the currently edited window.



## Paste Window Bitmap to Clipboard

Copy an image of the current layout to the Clipboard as a bitmap.

## Test Window

Launch 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. CompositePanels will be launched into a WBCompositePaneTester window.



# View

(obsolete)

The use of multi-viewed ViewManagers is discouraged. If you wish to utilize this capability, enable the “ViewManager Multi-View Support” add-in in the Add-In Manager.

## Switch To...

Prompts the user with a list of the other views in this viewmanager to edit. After choosing one, WindowBuilder Pro will display that view. If the viewmanager has not yet been saved to a class name, if there is only one view, or if you are currently editing a windowdialog, this command is dimmed.

## Create...

Prompts the user for the name of a new view to add to the window. After giving the new name, WindowBuilder Pro will switch to the new view, using the default template.

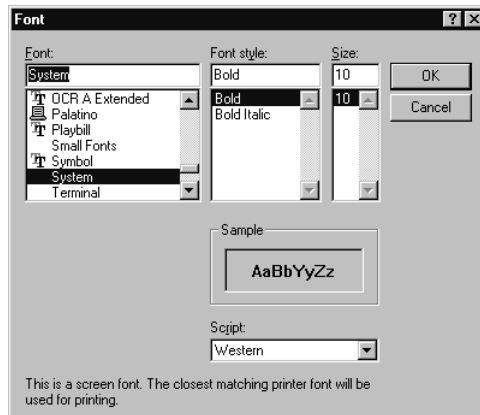
## Remove...

Removes the current view from this viewmanager. If there is only one view, this command will be dimmed.

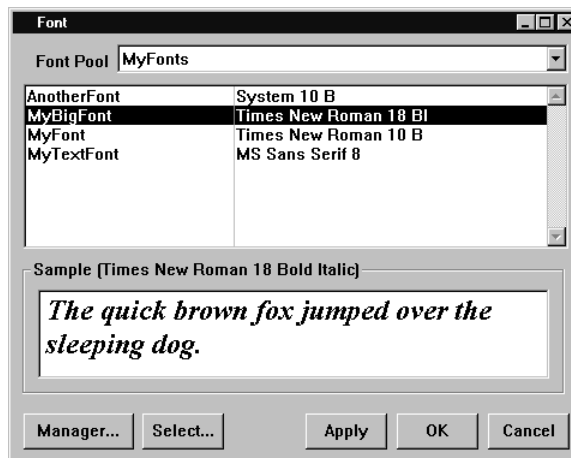
# Attributes

## Font...

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

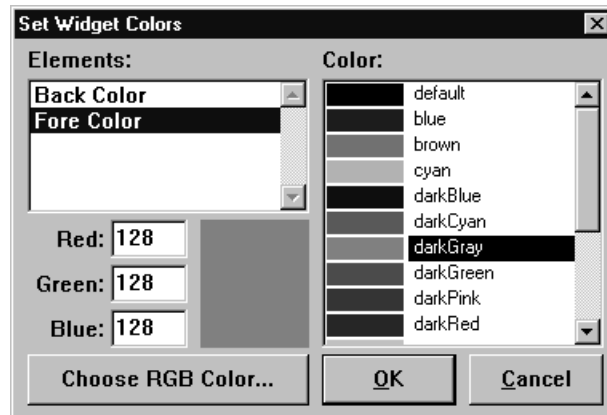


If you are using WindowBuilder Pro's advanced font pool management capabilities, you will see the font dialog shown below. This dialog allows you to select a pre-defined named font from an existing font pool. The standard font editor may be launched from this dialog by clicking on the Select button. If you are using font pool management and you want to select a font directly from the standard font dialog, you may do so by holding the ALT key down when clicking the Font button.



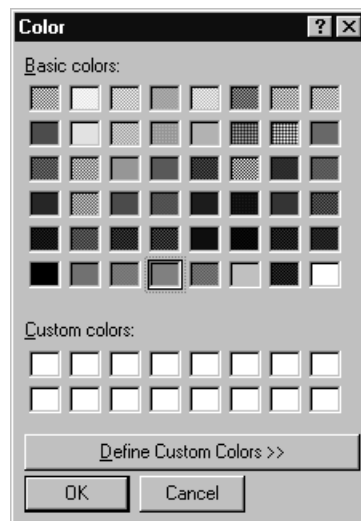
**Color...**

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

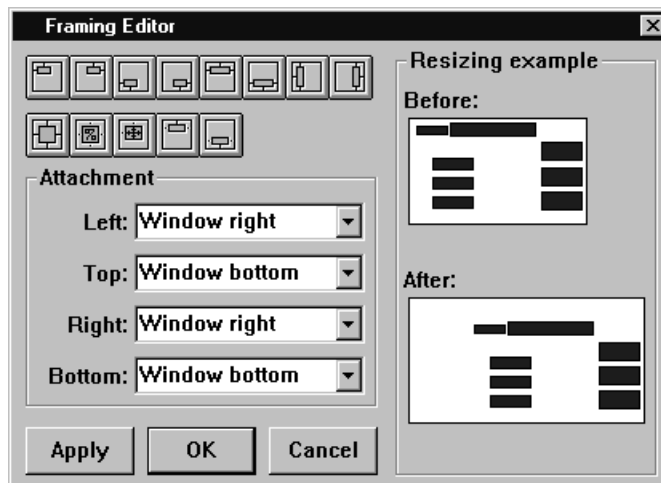


The first item in the color list will always be “default”. Selecting this color will set the widget to use its own default colors for that attribute. When a widget has been defined to use its own default color, no color attribute code will be generated.

The system color editor can be invoked by clicking on the Choose RGB Color button. It is shown below:

**Framing...**

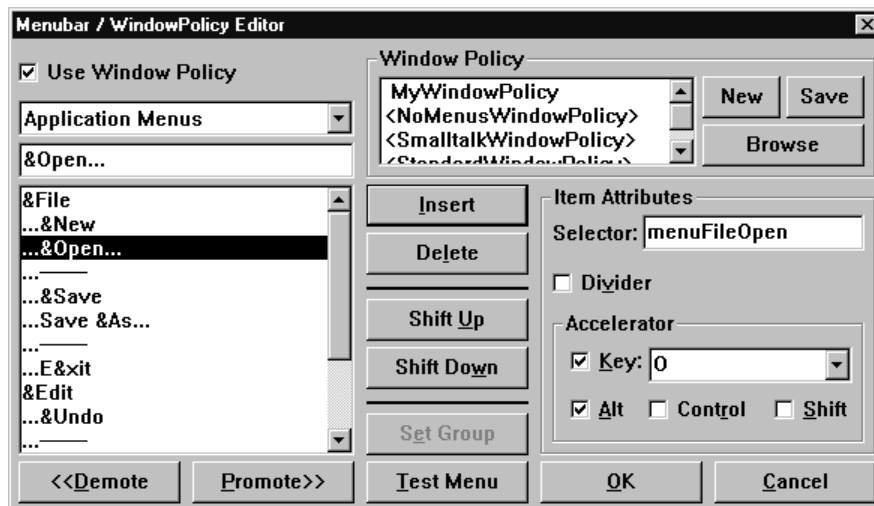
Displays an Framing Editor for the currently selected widget.



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 proportional attachments or fixed size attachments. A comprehensive list of attachment styles are provided via the toolbar at the top of the screen. By default the widget is constrained with respect to the application window (its top left corner).

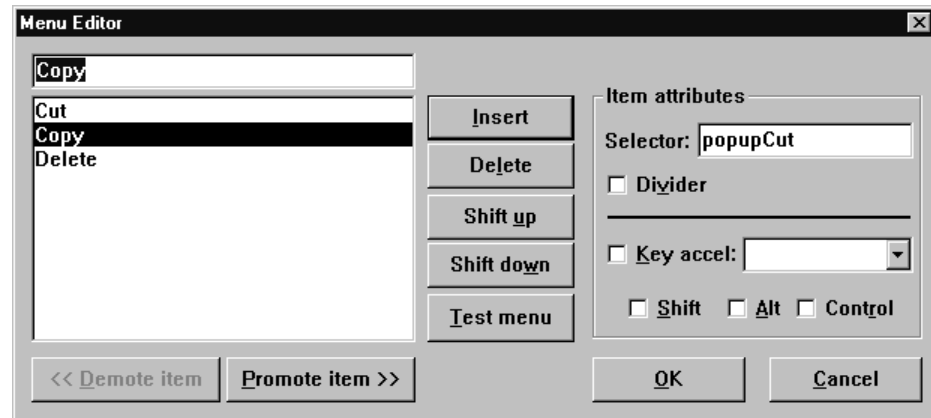
## Menus...

Displays a Menubar / WindowPolicy Editor for the window.



The Menubar / WindowPolicy WindowPolicy Editor is used to defined the application's pull down 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. Menu definitions may be stored with the class itself or may be saved as separate WindowPolicy WindowPolicy subclasses. WindowPolicies provide a convenient mechanism for sharing menu definitions between multiple windows.



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:

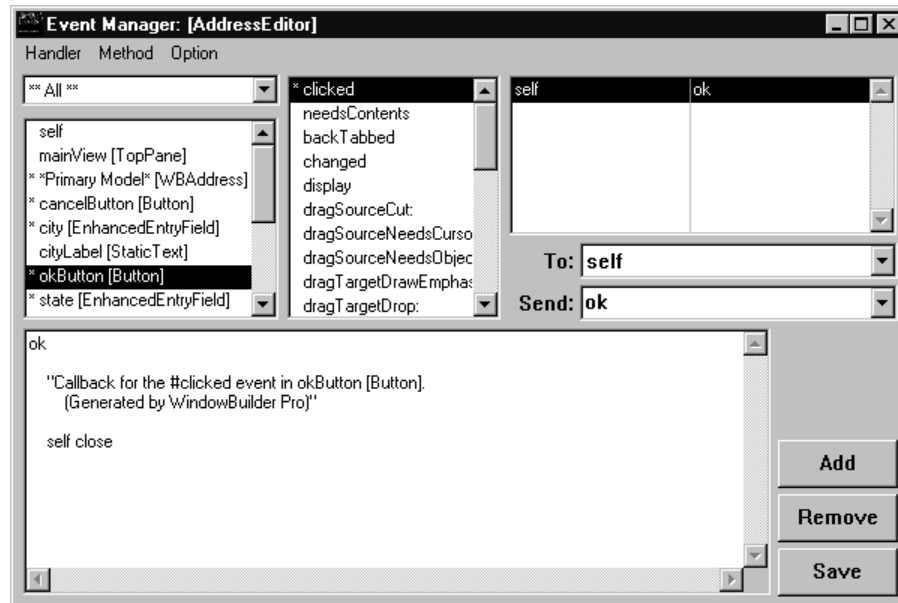
- When you create a menubar, the top level menu items are menu titles that display horizontally on the menubar. When you create a popup menu, the top level menu items display vertically when the user right-clicks on the widget associated with the menu.
- When you test a menubar, a new window launches 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

## Events...

Displays the Event Manager. The Event Manager window creates, edits, and removes event links between objects. Any window, widget, model, or global that triggers events can be linked to a method of any other window, widget, model, or global. For example, using the Event Manager to create a link between the `#clicked:` event of a list box and a method called `#myListBoxClicked:`, when the user selects an item in that list box, the `#clicked:` event is triggered, and a message is sent to the `#myListBoxClicked:` method where processing occurs to handle the user's selection. In addition, you can create and edit the event handlers (the `#myListBoxClicked:` method) in the Event Manager window.

In the Event Manager window, the object list (left top of the Event Manager window) displays the objects that trigger events. If the object has links to one or more of its events, then an asterisk appears preceding the name of the object. Directly above the object list is a combo box used for filtering the list of objects.

The event list (center top of the Event Manager window) contains the events for those objects selected in the object list. The events appear in alphabetical order.



The handler list (right top of the Event Manager window) contains the handlers defined for the objects selected in the object list and the events selected in the event list. If one object and one event is selected, then the handler list displays the target object (the object receiving the message) and the target selector (the message sent to the target object when the event is triggered). If multiple objects in the object list are selected, then the handler list displays the source object (the object triggering the event) in addition to the target object and the target selector. If multiple events are selected, then the handler list displays the event being triggered in addition to the target object and the target selector.

Below the handler list appears the target object drop down list and the target selector combo box. If no handler is selected in the handler list, then selecting a target object in the target object drop down list and entering a target selector in the target selector combo box will add a new handler to the handler list for the selected source object and event. If a handler is selected in the handler list, then selecting a target object or changing the target selector will change the target object and/or target selector for that selected handler.

To enter a new handler when no handlers are selected, select the target object and enter the target selector in the target object drop down list and target selector comb box respectively. To enter a new handler when one or more handlers are selected, select Add from the Handler menu, then select a target object and enter a target selector. To edit a handler, select the handler and then alter the target object and/or target selector. To remove a handler, select the handler then select Remove from the Handler menu.

The Event Managers's menus are described below:

## **Handler**

### **Add**

Deselects the currently selected handler (if one is selected) and positions the cursor in the target selector field so that when a target selector is entered, a handler with the selected target object and target selector will be added to the handler list for the currently selected source object and event.

### **Remove**

Removes the currently selected handlers.

### **Top**

Moves the currently selected handler(s) to the top of the handler list. This only works if there is exactly one source object selected and exactly one event selected.

### **Up**

Moves the currently selected handler(s) up one position in the handler list. This only works if there is exactly one source object selected and exactly one event selected

### **Down**

Moves the currently selected handler(s) down one position in the handler list. This only works if there is exactly one source object selected and exactly one event selected

### **Bottom**

Moves the currently selected handler(s) to the bottom of the handler list. This only works if there is exactly one source object selected and exactly one event selected

**Show All**

Expands the handler list to fill the entire upper half of the Event Manager window and displays the source object, event, target object, and target selector for all defined handlers.

**Method****Cut**

Cut the selected text in the text edit area to the clipboard.

**Copy**

Copy the selected text in the text edit area to the clipboard.

**Paste**

Paste the clipboard text into the text edit area.

**Clear**

Clear the selected text from the text edit area without affecting the clipboard.

**Do It**

Evaluate the selected text in the text edit area.

**Show It**

Evaluate the selected text in the text edit area and display the result.

**Inspect It**

Evaluate the selected text in the text edit area and open an Inspector on the result.

**Save**

If the method source in the text edit area has been modified, this will save the method source.



## Option

### Show All Selectors

If selected, then all selectors for all currently defined method for the currently selected target object will appear in the target selector drop down list. If not selected, then only the selectors for those methods in classes up to the class inheriting object will be shown in the target selector drop down list.

### Show All Buttons

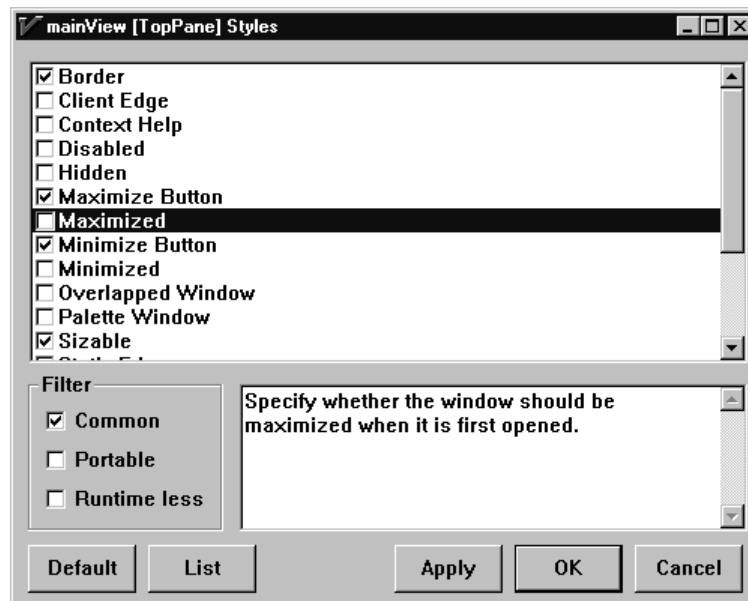
If selected, then the add, remove, and save buttons to the right side of the text area will be displayed.

### Drag Drop Handlers

If selected, then the handlers may be sequenced using normal drag and drop techniques.

## Styles...

Displays the Style Editor. The Style Editor is useful for setting a widget or window's boolean properties or operating system specific style attributes. The list of styles may be filtered to include only Common styles (as in commonly used), Portable styles, and/or Runtime less styles.



## Attributes...

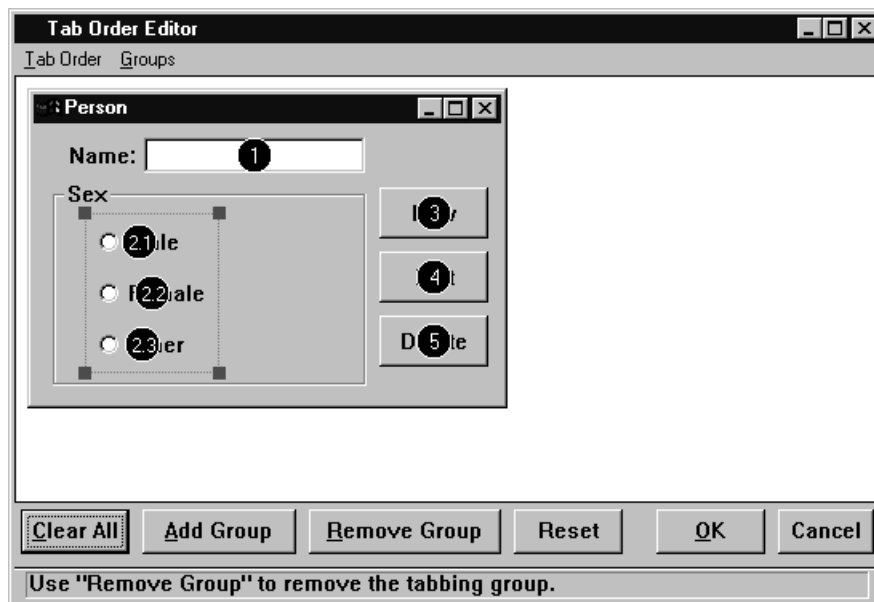
Displays the attribute editor for the currently selected window or widget(s). Each attribute editor is described in the Widget Encyclopedia in conjunction with the widget type it edits.

## Tabbing/Groups...

Displays the Tab Order Editor. This editor displays any widgets you've laid out with WindowBuilder Pro. If you have already entered a tab order for the widgets, numeric labels representing this order are displayed on each widget. If you have not yet set the tab order, no labels are displayed on the widgets.

Click the widgets in the order in which you want them to be selected with the TAB key. A numeric label appears on each widget when you click it, representing its position in the tab order. If you click a widget that does not make use of the input focus (such as a `StaticText` widget), the system beeps, and no number will appear on the widget.

If you make a mistake, simply ALT click on a widget to remove its tab stop. To swap two widgets in the tab order, click on the first widget and drag its tab stop number to the second widget. This drag drop capability makes it easy to rearrange the tab order without having to start over again.



To create tab group, press the Add Group button and rubberband-select the widgets that you want to group. A rectangular box appears around them, indicating that they're now a group. In addition, the previous tab order will be readjusted, to indicate that these components are part of a group.

The Tab Order Editor's menus are described below:

## **Tab Order**

### **Smart Set**

Establish a default tab order based on the layout of the window. The default tab order may be establish in either column or row major order (or based on the current z-order in the window). These options use an intelligent algorithm that will automatically identify nested widget groups.

### **Clear All**

Clears the entire tab order.

### **Reset**

Resets the tab order to its initial point when the window opened.

### **Reverse**

Reverse the tab order for the entire window. The order of widgets within a tab group are not reversed. To reverse the order of widgets in a tab group, select the group and then select Reverse.

### **Exit**

Closes the Tab Order Editor and saves any changes.

## **Groups**

### **Add**

Add a tab group to the window.

### **Remove**

Remove a tab group from the window.

## Create All

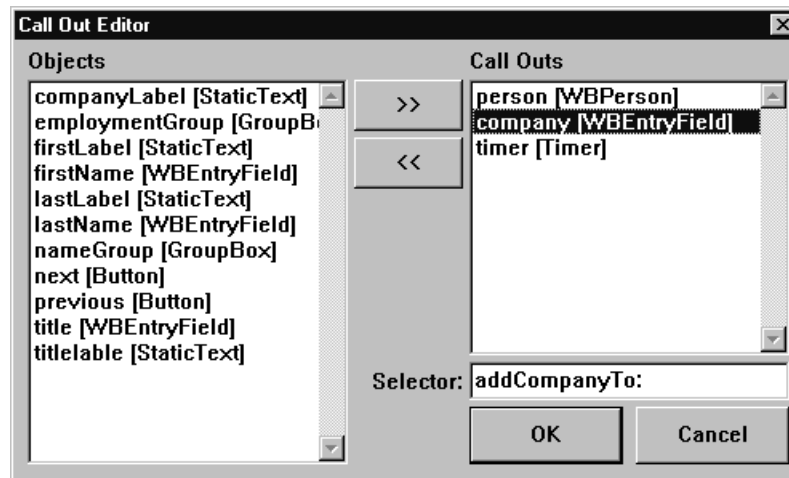
Create default tab groups based on the layout of the window. This takes into account nested GroupBoxes.

## Include Outer

This determines whether the system will create a group for any buttons that are outside of any GroupBox when either the Smart Set of Group | Create All options are used.

## Call Outs...

Displays the Call Out Editor.



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 `#createViews` method (which can cause problems with the VisualSmalltalk compiler). The Call Out Editor allows you to have any widget or model object generated into its own method. The editor interface is simple. On the left side of the window is a list of all widgets and model objects. The `>>` button moves one or more objects to the right hand list where a call out method selector may be specified. Note that this should be one-argument selector and that a default method name is built for you based on the object's name. The `<<` button may be used to transfer an object back to the left hand list where it will be generated in the main `#createViews` method.

## NLS

Displays a submenu with the following choices:

### Set Pools...

Displays the NLS pool assignment window.

Text labels for buttons, labels, window titles and menus can be specified as NLS pool dictionary keys rather than strings. Use this dialog to assign one or more NLS pool dictionaries to the currently edited class.



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 the Text field 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.

### NLS Manager...

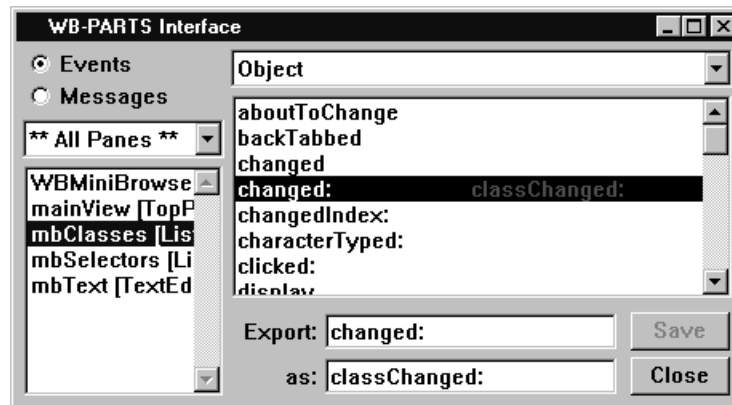
Launches the NLS Manager window. See the *Pool Managers* chapter for complete details.

### NLS Autosize

Specifies whether autosize operations should be NLS-aware. If this option is turned on, a widget which is using an NLS key as a label will be autosized based on the longest string mapping for that key across all of the defined language categories.

## PARTS Interface...

Displays the WB-PARTS Interface window. Windows generated by WindowBuilder Pro can be used in any PARTS application. After building the window using WindowBuilder Pro, open this window to define which events and messages are visible to the PARTS programmer. Once the PARTS interface is defined, you may drag the PARTS icon in the WindowBuilder Pro layout window and drop it in the PARTS Workbench.



The lower left pane in the WB-PARTS Interface window contains a list of names of all widgets that you have defined. The drop down list above it contains filters for this list. You may show all widgets or only those widgets of a particular class.

Selecting a widget name displays a series of events or messages in the right hand pane. The drop down list above it contains filters for this list. You may show all events or messages, or only those events or messages defined in a particular class.

Each of these events or messages may be exported as part of the WB-PARTS interface for your window. Selecting an event or message in the right hand pane displays the original name of the event or message in the Export: entryfield. To specify that an event or message be exported, type the new name of the event or message in the As: entryfield. This new name will be used whenever linking events or messages in the PARTS Workbench.

# Align

## Left

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

## Top

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

## Right

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

## Bottom

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

## Center Horizontally

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

## Center Vertically

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

## Distribute Horizontally

Evenly distribute the selected widgets horizontally, i.e., leaving the first and last widgets in the same location, force the space between each widget to be the same.

## Distribute Vertically

Evenly distribute the selected widgets vertically, i.e., leaving the first and last widgets in the same location, force the space between each widget to be the same.

**Note:** 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 they were selected

## Move By Pixel

Move the selection one pixel in the direction specified.

### **Size By Pixel**

Size the selection one pixel in the direction specified. When sizing, the top left corner of the selection will remain stationary, and the lower right will shift.



# Size

## Auto Size Selection

Set the size of the selected widget to the size specified by the widget's answer to the `#suggestedSize` message. The default suggested size is the current size of the widget, i.e., a no-op. This command is useful for “sanely” sizing simple widgets like static text without much effort.

## Replicate Width

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

## Replicate Height

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

## Set Window Size...

Set the size of the selected widget or window to a specific pixel size. This is useful for giving windows well-known sizes, such as the size of a VGA screen.

## Set Window Position...

Set the location at which the currently selected widget or window will initially be placed. For a window, this command will track the mouse with a rectangle the size of the current window; when the user releases the mouse, the current position of the rectangle will be used. For a widget, a standard prompter dialog will appear.

# Options

## Grid

Displays a submenu with the following choices:

### Use Grid

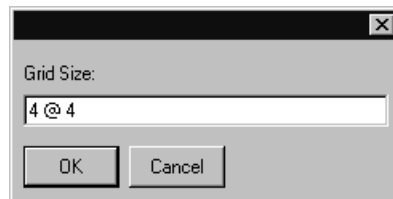
Toggles the grid function on or off. When set on, an invisible grid is overlaid on the design surface that widgets snap to when being placed, dragged, or sized.

### Draw Grid

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

### Set Grid Size...

Set the grid size of the layout pane in the x and y directions. Grid size is useful for aligning widgets; it creates an invisible grid that widgets “snap” to when being placed, dragged, or sized.



## Look Policy

Displays a submenu with the following choices:

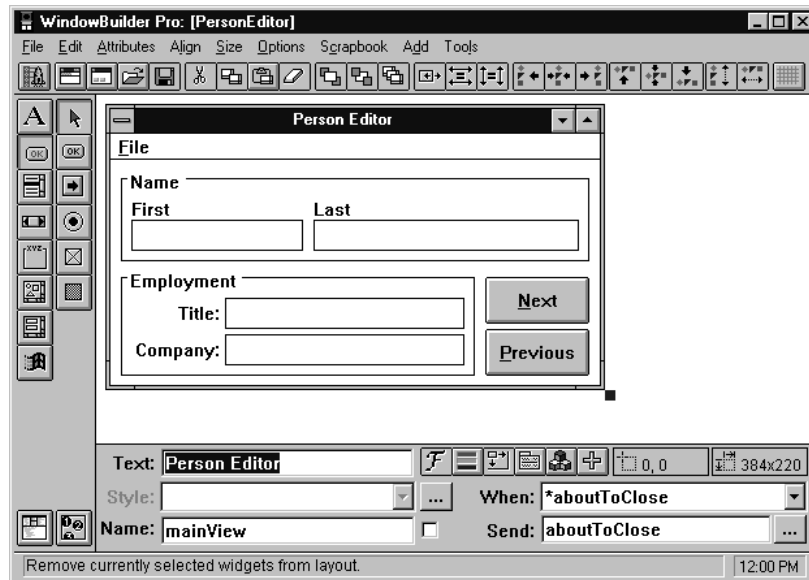
### Default

Displays the window using the default platform look and feel. For example, under Windows 95, the window and widgets would be displayed using the Windows 95 look and feel.

**Note:** These look policies only apply to the view of the window within the editor and have no effect on the live running version of the window.

## Windows 3.1

Displays the window using the Windows 3.1 look and feel.



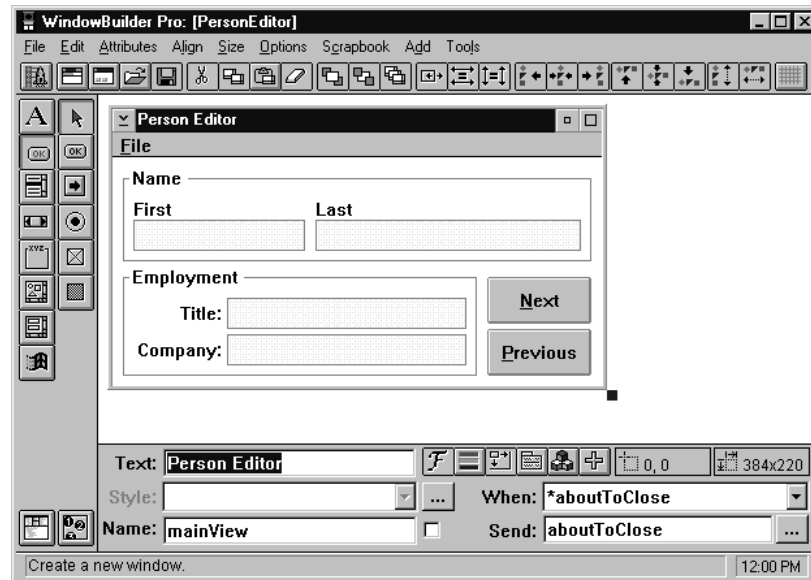
## Windows 95

Displays the window using the Windows 95 look and feel.



## OS/2

Displays the window using the OS/2 look and feel.

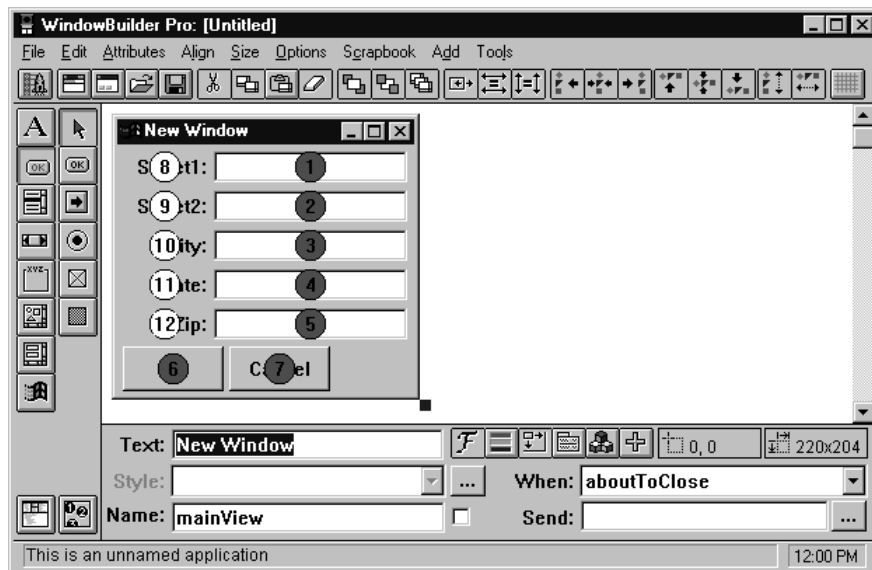


## Drag Outlines

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 Z-Order

Toggles the Z-order display function on and off. When set on, a filled circle containing a widget's 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 tab stops are indicated by red. Widgets that are not tab stops at all are white (e.g., labels and other static elements).



## Target Is First

Toggles the selection mode between target-is-first mode and target-is-last mode. 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 window. Turning this off will allow a widget to be positioned anywhere within its parent (even off screen out of view).

## Update Outboards

Toggles the outboard update function on and off. When set on, WindowBuilder Pro will automatically update any outboard windows (e.g., framing 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 promotes more rapid testing.

## Auto Size

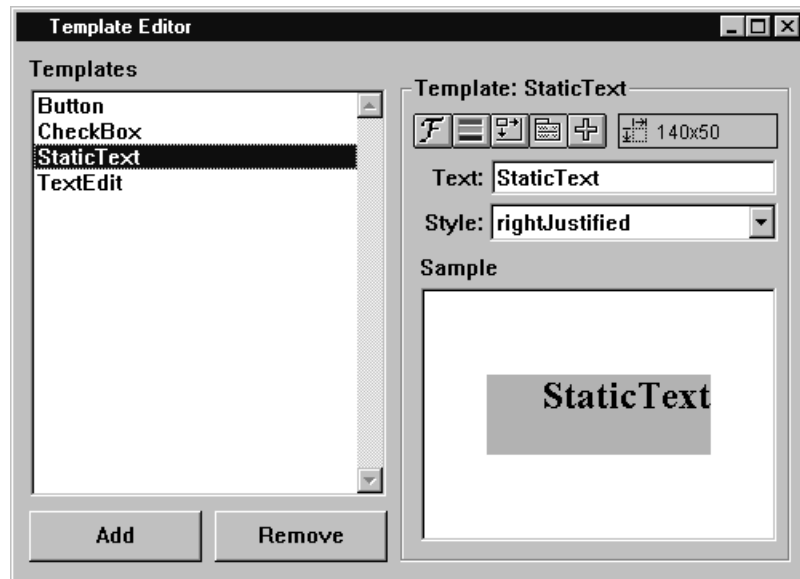
Toggle the state of automatic autosizing. If autosizing is on, many widgets (StaticText, Buttons, etc.) will automatically adjust their sizes as you type their labels. StaticText widgets will autosize based on their justification style.

## Zoom Layout

Zoom the layout area of WindowBuilder Pro to take the entire window. This command toggles; executing it again restores the full window.

## Templates...

Displays the Template Editor.

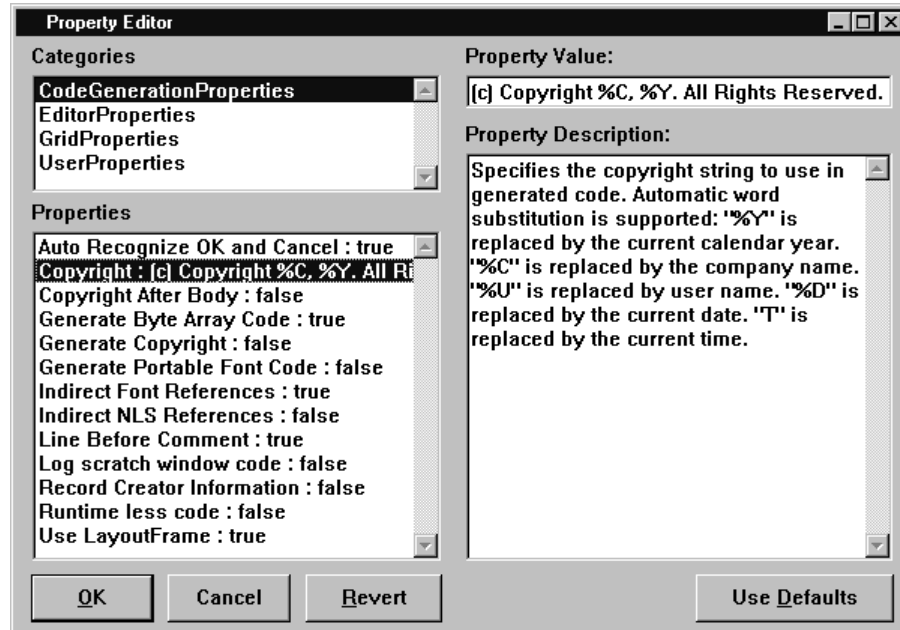


The template editor allows you to specify default attribute values for attributes of all the widgets supported by WindowBuilder Pro. Once you change a default value, that new value will be used for all new instances of that widget that you place in the editor. For example, all StaticText widget are left justified by default. This can easily be changed so that all new labels are right justified when first dropped into the editor.

## Properties...

Displays the Property Editor.

The property 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.



<b>Auto Recognize OK and Cancel</b>	Specifies whether Buttons with labels “OK” or “Cancel” should automatically be given the defaultPushButton or cancelButton style during code generation.
<b>Auto Save</b>	Specifies whether the window definition should automatically be saved before testing a window.
<b>Auto Size</b>	Specifies whether widgets should automatically auto size when their contents are changed within the editor.
<b>Auto Update Outboards</b>	Specifies whether the outboard windows should be updated when new widgets are selected in the main editing window.
<b>Cache Property Managers</b>	Specifies whether the property managers for widget classes should be cached or rebuilt each time as needed.
<b>Comment Methods</b>	Specified whether the comment field of each method should be set to the current copyright string.

<b>Company Name</b>	Specifies the user's company name.
<b>Copyright</b>	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.
<b>Copyright After Body</b>	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.
<b>Drag Outlines</b>	Specifies whether widgets should be dragged as outlines or not.
<b>Draw Grid</b>	Specifies whether the grid is visible within WindowBuilder Pro.
<b>Generate Byte Array Code</b>	Specifies whether byte arrays should be generated using the new language specification. If true, then ByteArrays will be generated as  <code>#[ ]</code>  rather than  <code>#( ) asByteArray</code>
<b>Generate Copyright</b>	Specifies whether each generated method should include a copyright string.
<b>Generate Portable Font Code</b>	Specifies whether font definitions should be generated using portable font APIs. Non-portable font generation is more accurate on the current platform.
<b>Generate SysFont References</b>	Specifies whether font references to the system font ( <code>font : SysFont</code> ) should be generated.
<b>Grid Size</b>	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.
<b>Handle Size</b>	Specifies the handle size used in the WindowBuilder Pro editor.



<b>Indirect Bitmap References</b>	<p>Specifies whether references to bitmaps managed by the Bitmap Manager should be generated as indirect references</p> <p>Bitmap named: &lt;key&gt;</p> <p>or direct references</p> <p>&lt;key&gt;</p>
<b>Indirect Font References</b>	<p>Specifies whether references to fonts managed by the Font Manager should be generated as indirect references</p> <p>Font named: &lt;key&gt;</p> <p>or direct references</p> <p>&lt;key&gt;</p>
<b>Indirect NLS References</b>	<p>Specifies whether references to strings managed by the NLS Manager should be generated as indirect references</p> <p>&lt;NLS Pool&gt; named: &lt;key&gt;</p> <p>or direct references</p> <p>&lt;key&gt;</p>
<b>Line Before Comment</b>	<p>Specifies whether there should be a blank line between the message pattern and the method comment.</p>
<b>Log Scratch Window Code</b>	<p>Specifies whether code generated for the WindowBuilder scratch windows should be logged.</p>
<b>Make Callbacks Private</b>	<p>Specifies whether generated callback stubs should be private methods. True indicates that they should be private. False indicates that they should be public.</p>
<b>Max Undo Levels</b>	<p>Specifies the maximum number of undo levels maintained by the WindowBuilder Pro editor.</p>
<b>Max Window Size</b>	<p>Specifies the maximum window size that can be built using WindowBuilder Pro.</p>
<b>NLS Autosize</b>	<p>Specifies whether the target widget is auto-sized based on the widest NLS string.</p>
<b>PARTS Support</b>	<p>Specifies whether PARTS Support should be enabled or not.</p>

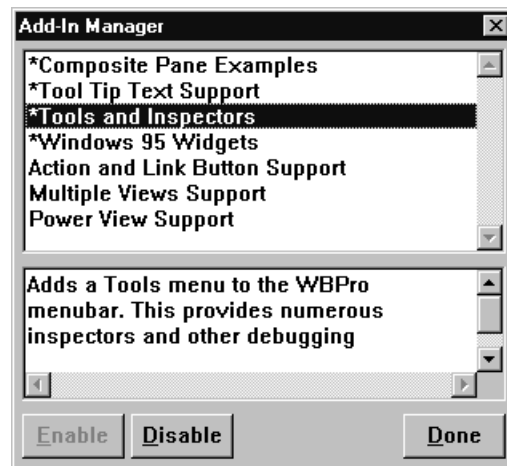
<b>Record Creator Information</b>	Specifies whether creator information should be generated.
<b>Runtime Less Code</b>	Specifies whether generated code should not have any dependencies on WindowBuilder Pro runtime libraries.
<b>Serial Number</b>	Specifies the user's serial number.
<b>Show Z Order</b>	Specifies whether the z-order of the widgets should be shown in the editor.
<b>Target Is First</b>	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.
<b>Use ClassHierarchy Browser</b>	Specifies whether the "Browse Class..." menu item should open the ClassHierarchyBrowser or the platform specific class browser (PackageBrowser for Team/V).
<b>Use Fence</b>	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).
<b>Use Grid</b>	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.
<b>Use LayoutFrame</b>	Specifies whether LayoutFrame or FramingParameters should be used during code generation.
<b>Use WBComboBox</b>	Specifies whether WBComboBox should be used by default rather than ComboBox.
<b>User Name</b>	Specifies the user's name.

## Manager

Displays a submenu with the following choices:

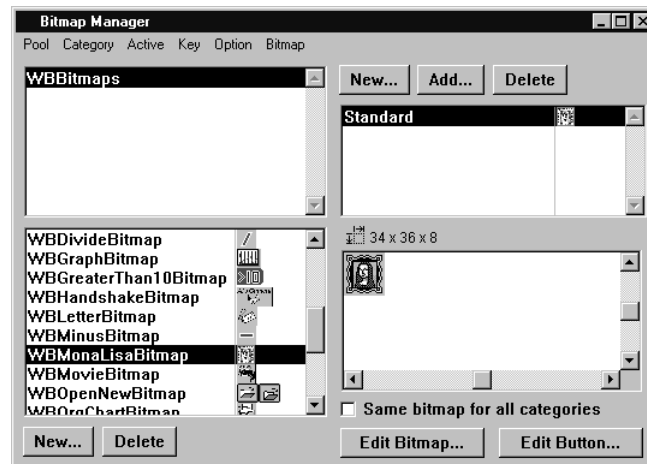
### Add-In Manager

Launch the Add-In Manager dialog. The Add-In Manager is used to extend the WindowBuilder Pro environment with additional functionality.



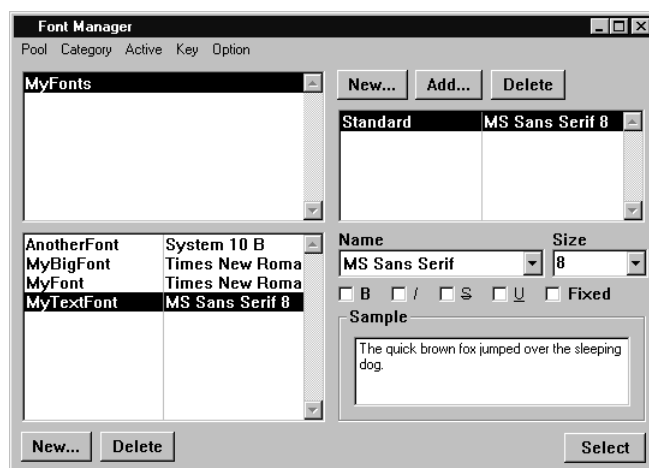
## Bitmap Manager

Launch the Bitmap Manager window. The Bitmap Manager is used to manage bitmaps and bitmap pool dictionaries. See the *Pool Managers* chapter for complete details.



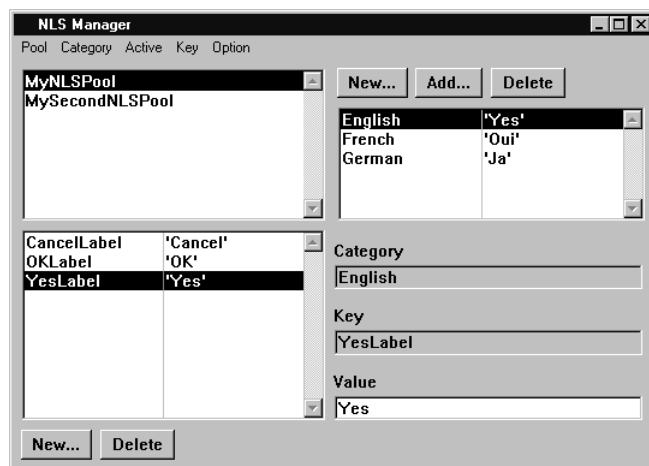
## Font Manager

Launch the Font Manager window. The Font Manager is used to manage fonts and font pool dictionaries. See the *Pool Managers* chapter for complete details.



## NLS Manager

Launch the NLS Manager window. The NLS Manager is used to manage NLS string dictionaries and language mappings. See the *Pool Managers* chapter for complete details.



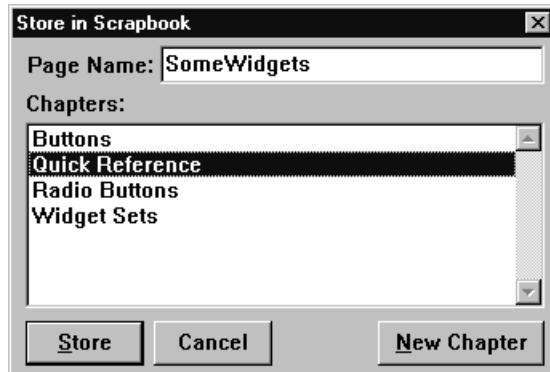
## Redraw

Forces the window to repaint itself.

# Scrapbook

## Store..

Store the selected widget or widgets in the Scrapbook for later retrieval.

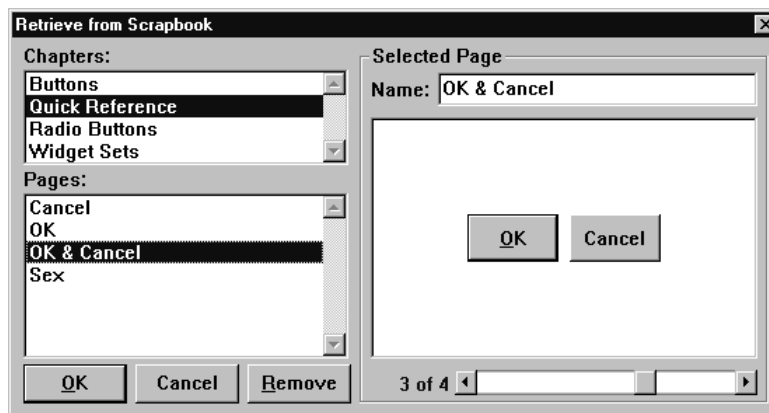


The Page Name entryfield is used to record the name of the new page. The listbox below it shows all of the chapters that have been defined. New chapters may be easily added by clicking on the New Chapter button. You may select as many chapters as you like in which to store the new page.

**Note:** The *Quick Reference* chapter is special. Any pages you put in it will appear as cascaded menu items on the Scrapbook | Quick Reference menu.

## Retrieve..

Open the Scrapbook and select a stored item for use in the editing window.



The Chapters listbox lists all of the defined chapters. The Pages listbox below it shows all of the pages that make up the chapter. You can scroll through the list of pages by using the scrollbar located at the bottom right of the dialog, or by clicking the names of the pages in the Pages list box.

### **New**

Create a new Scrapbook. If you have not saved the current Scrapbook to disk, its contents will be lost.

### **Load..**

Load a Scrapbook from disk (.SBK files). This will replace the current Scrapbook in its entirety.

### **Merge..**

Merge a Scrapbook stored on disk with the one in your image. Conflicts (pages with the same name in the same chapter) can be resolved by keeping the current version, using the new version, or bringing the new version in with a “.2” appended to its name.

### **Save..**

Save the current Scrapbook to disk.

### **Quick Reference**

Cascade a menu showing all of the pages contained in the “Quick Reference” chapter of the Scrapbook. This chapter is an ideal place to put standard interface objects that are used frequently (for example, the “OK & Cancel” combination).

# Add

## Text

Drops down a submenu containing all the text display and editing widgets. 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 Widget Palette).

## Button

Drops down a submenu containing all the button-type widgets. It acts like Text, above.

## List

Drops down a submenu containing all the “choice pane”-type widgets, or widgets which allow the user to select one from a group of items. It acts like Text, above.

## Valuator

Drops down a submenu containing all the “valuator”-type widgets, or widgets which allow the user to set a numeric value from a range. In the base system, this consists only of scrollbars. It acts like Text, above.

## Group

Drops down a submenu containing all the visually containing widgets including GroupBoxes and StaticBoxes. It acts like Text, above.

## Misc

Drops down a submenu containing a miscellaneous group of widgets, including GraphPane, AnimationPane, and WBToolBar. It acts like Text, above.

## Composite

Drops down a submenu containing special CompositePane subclasses, including RadioButtonGroup, CheckBoxGroup, and EntryFieldGroup. It acts like Text, above.

**Note:** CompositePanes that you create will not appear on this menu unless you add them using the procedure described in *Appendix A Customizing WindowBuilder Pro*. Use the **Custom Widgets** menu below.

## Windows 95

**(Windows only)**

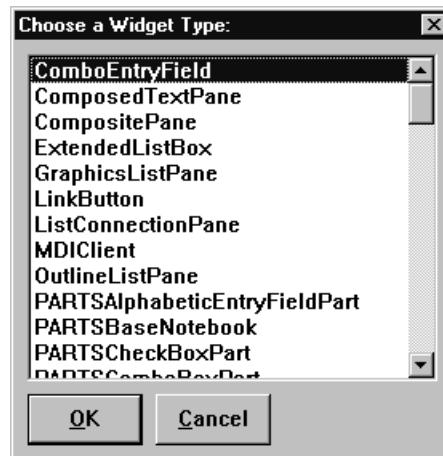
Drops down a submenu containing all the Windows 95 widgets. It acts like Text, above.

## Custom Widgets

Drops down a submenu that allows you to place a custom widget of your design (or any other widget that is not on the WindowBuilder Pro palette). Widgets can be added to this list using the Add Custom Widget... command, or removed using the Remove Custom Widget... command, both described below.

### Add Custom Widget...

Adds a widget to the Custom Widgets list. If you create your own widgets, and want access to them in WindowBuilder Pro, add them using this command. You may also add them to the WindowBuilder Pro palette by following the instructions in *Appendix A Customizing WindowBuilder Pro*.



### Remove Custom Widget...

Removes a widget from the Custom Widgets list.





# Chapter 10 Widget Encyclopedia

This section of the manual provides a reference description of the widgets that can be found on WindowBuilder Pro's widget palette, in alphabetical order. We have also provided a reference description of the **SubPane** and **ControlPane** classes for convenience. Each description includes the following:

- the source of the widget: Objectshare or ParcPlace-Digitalk
- an overview of the widget's functionality and general usage
- a list of the public protocol for the widget
- a list of the supported events for the widget
- a description of any WindowBuilder Pro extensions associated with the widget

## All Widgets

### **AnimationPane**

Used to implement simple animation.

### **Button**

A simple push button.

### **ButtonListBox**

Provides the ability to toggle a list of choices on and off. Also provides single select listbox capabilities.

### **CheckBox**

A toggle button used to represent a true or false state.

### **CheckBoxGroup**

A CompositePane used to quickly implement groups of CheckBoxes.

### **ComboBox**

A hybrid between an EntryField and a ListBox allowing for text entry or list selection.

### **DrawnButton**

A simple push button that may contain a graphic rather than a string.

**DropDownList**

A widget providing popup list capability.

**EnhancedEntryField**

An advanced single line entry control that provides character and field level validation.

**EntryField**

A simple single line text entry control.

**EntryFieldGroup**

A CompositePane used to quickly implement groups of EntryFields.

**GraphPane**

Provides for general drawing within a pane.

**GroupBox**

Used to visually group related controls.

**Header**

(Windows 95 only)

Provides a simple column header control.

**ListBox**

Provides a general selection capability from a collection of mutually exclusive choices.

**ListPane**

A ListBox implemented entirely within Smalltalk.

**ListView**

(Windows 95 only)

An iconic container widget with multi-column list capabilities.

**MultipleSelectListBox**

A ListBox allowing selection of multiple choices within a list.

**ProgressBar**

(Windows 95 only)

A simple visual gauge.

**RadioButton**

A toggle button used to provide selection from mutually exclusive choices.

**RadioButtonGroup**

A CompositePane used to quickly implement groups of RadioButtons.

**RichEdit****(Windows 95 only)**

A multi-line Rich Text (RTF) editor that supports multiple fonts, colors and formatting options.

**ScrollBar**

A generic slider control.

**SexPane**

A CompositePane used to specify the sex of an individual.

**SpinButton****(Windows 95 only)**

Provides a general selection capability from a collection of mutually exclusive choices. Used as an alternative to ListBoxes when space is at a premium.

**StaticBox**

Used to draw empty or filled rectangles.

**StaticText**

A static pane used for displaying text.

**StatusPane**

Simple status display control that provides multiple status boxes at the bottom of a window.

**StatusWindow****(Windows 95 only)**

Provides multiple status boxes at the top or bottom of a window. Multiple box styles are supported.

**TabControl****(Windows 95 only)**

A multi-page notebook control with tabs.

**TextEdit**

A multi-line text editor.

**TextPane**

A multi-line text editor implemented entirely within Smalltalk.

**ThreeStateButton**

A CheckBox with a third state to reflect ambiguity.

**TrackBar****(Windows 95 only)**

A slider control supporting multiple scales and tick marks.

**TreeView** (Windows 95 only)

A hierarchical list control with expand and collapse capabilities.

**UpDown** (Windows 95 only)

A simple control that is a composite between an up and a down button.

**VideoPane** (Windows 95 only)

A simple multimedia control.

**WBStaticGraphic**

Used to display a static graphic.

**WBToolBar**

Used to implement standard horizontal toolbars.

# SubPane & ControlPane

*ParcPlace-Digitalk*

SubPane and ControlPane are the ultimate abstract superclasses of all widgets. They provides general behavior that applies to all of its subclasses. Before discussing each of the widgets in detail it is important to touch on them first.

## Protocol

### **abortChange**

Veto the change which is about to occur.

### **addBorderStyle**

Add the border style to the receiver.

### **addClipchildrenStyle**

Add the clip children style to the receiver.

### **addClipsiblingsStyle**

Add the clip siblings style to the receiver.

### **addGroupStyle**

Add the group style.

### **addHorizontalScrollbarStyle**

Add the horizontal scrollbar style to the receiver.

### **addTabStopStyle**

Add the tab stop style.

### **addTransparentStyle**

Add a style that allows mouse clicks to pass through the receiver to panes below it.

### **addVerticalScrollbarStyle**

Add the vertical scrollbar style to the receiver.

### **altKeyInput:** *altCharacter* **from:** *aPane*

An alt-character was typed in *aPane* of the receiver. Search for a child mnemonic. Answer whether the mnemonic is handled.

**asParameter**

Answer the receiver in a form suitable for passing as a parameter to a host procedure call.

**backColor**

Answer the background color of the widget.

**backColor:** *aColor*

Set the background color of the receiver to *aColor*.

**boundingBox**

Answer the bounding box of the receiver.

**bringToTop**

Bring the widget to the top of its overlapping siblings and activate it.

**captureMouseInput**

Send all mouse input to the receiver until `clearMouseCapture` is executed.

**children**

Answer the collection of the receiver's children.

**childrenSize**

Answer the size of all descendents.

**clearMouseCapture**

End mouse capture.

**contents**

Answer the label of the receiver.

**contents:** *aString*

Set the String associated with this control.

**cursorWindowPosition**

Answer the current position of the cursor in window coordinates.

**defaultBackColor**

Answer the default background color of the receiver.

**defaultCursor**

Answer the default cursor for the receiver.

**defaultFont**

Answer the default font of the receiver.

**defaultForeColor**

Answer the default foreground color of the receiver.

**disable**

Disable the widget.

**disabled**

Answer true if the receiver is disabled, else false.

**disableRedraw**

Disable redrawing of the receiver.

**disableUpdate**

Disable update of the receiver and all its children.

**dragSessionClass:** *aClass*

Set the class to use to model a drag drop transfer to *aClass*.

**dragSource**

Enable the receiver as a drag drop source.

**dragSource:** *aBoolean*

Enable or disable the receiver as a drag drop source.

**dragTarget**

Enable the receiver as a drag drop target.

**dragTarget:** *aBoolean*

Enable or disable the receiver as a drag drop target.

**dragTargetEmphasisDefault**

Set the type of target emphasis to be employed to be the default.

**dragTargetEmphasisItem**

Set the type of target emphasis to be employed to be item based.

**dragTargetEmphasisPane**

Set the type of target emphasis to be employed to be pane based.



**dragTargetEmphasisSeparator**

Set the type of target emphasis to be employed to be separator based.

**dragTargetForFormats:** *formats* **operations:** *operations*

Enable the receiver as a drag drop target which accepts the specified DragDropObject formats and drop operations. If this method is used to setup a drag target (instead of #dragTarget), the #dragEnter: event can be left unhandled and a default handler will be provided.

**dragTargetMultipleItem**

Answer whether the receiver will accept multiple item drag-drop transfers.

**dragTargetMultipleItem:** *aBoolean*

Determine whether the receiver will accept multiple item drag-drop transfers, according to the value of *aBoolean*.

**drawBox**

Answer the bounding box of the item to be drawn.

**drawIndex**

Answer the index of the item to be drawn.

**drawingRectangle**

Answer a rectangle which completely encloses the drawable area of the receiver (exclusive of scrollbars & border).

**enable**

Enable the widget.

**enableRedraw**

Allow the receiver to be redrawn; force the receiver to repaint itself.

**enableUpdate**

Enable update of the receiver and all its children.

**extent**

Answer the receiver's extent.

**font**

Answer the font of the widget.

**font:** *aFont*

Set the font of the widget to *aFont*.

**foreColor**

Answer the foreground color of the widget.

**foreColor:** *aColor*

Set the foreground color of the widget to *aColor*.

**frameRectangle**

Answer the receiver's frame area as a rectangle.

**frameRelativeRectangle**

Answer the receiver's frame window rectangle relative to the super window.

**frameWindow**

Answer the frame window of the receiver.

**framingBlock:** *aBlock*

Set the framing block of the widget to *aBlock* which, when executed, yields the pane frame rectangle. *aBlock* may either be a one argument block or, in the case of WindowBuilder Pro, an instance of a LayoutFrame object.

**framingRatio:** *aRectangle*

Set the framingBlock of the widget to a block which, when executed, yields the pane frame rectangle proportional with the ratios specified by *aRectangle*.

**getMnemonicHandler:** *char*

Answer an object to handle the mnemonic *char*. Default is to answer nil.

**getStyle**

Answer the initial style of the receiver, composed of its basicStyle and its defaultStyle.

**getValue**

Answer the value of the receiver. This should be reimplemented by the subclasses in order to get the up to date value.

**hasDescendant:** *aWindow*

Answer whether *aWindow* is a child window of the receiver.

**hasFocus**

Answer true if the receiver currently has the focus.

**hasStyle:** *aStyle*

Answer whether the receiver has the specified bits set in its style flag.

**hasTransparentStyle**

Answer whether the receiver has the transparent style.

**height**

Answer the receiver's height.

**hideWindow**

Make the widget invisible.

**id**

Answer the id of the receiver.

**invalidate**

Repaint the receiver

**invalidateRect:** *aRectangle*

Force the region of the widget specified by *aRectangle* to be repainted. If *aRectangle* is nil, the entire widget will be redrawn.

**invalidateRect:** *aRectangle* **erase:** *eraseBackground*

Invalidate *aRectangle* area in the receiver thus force the area to be repainted. If *eraseBackground* is true then the area is erased before being repainted.

**isActive**

Answer true if the receiver's main window is the currently active window.

**isDragButton:** *anInteger*

Answer whether the mouse button index identified by *anInteger* is a button that initiates a drag transfer on this host.

**isDragSource**

Answer whether the receiver is enabled as a drag drop source.

**isDragTarget**

Answer whether the receiver is enabled as a drag drop target.

**isGlobalDragSource**

Answer whether the receiver is enabled as a local drag drop source.

**isHandleOk**

Answer whether the host window exists (has been created).

**isHidden**

Answer whether the receiver is hidden.

**isLocalDragSource**

Answer whether the receiver is enabled as a local drag drop source.

**isOffScreen**

Answer true if any part of the receiver is off the screen.

**isValid**

Answer whether the host window exists (has been created).

**isVisible**

Answer true if the window is visible

**label**

Answer the receiver's label

**layoutFrame**

Answer the receiver's layout frame.

**layoutFrame:** *aLayoutFrame*

Set the receiver's layout frame to *aLayoutFrame*.

**listFont:** *aFont*

Set the list pane font of all children to *aFont*.

**mainWindow**

Answer the receiver's main window. For example, if the receiver is a subpane, the main window will be an instance of TopPane.

**mnemonic:** *char* **typedIn:** *aPane*

The receiver's mnemonic was typed in the sibling *aPane*. Perform appropriate behavior. Default is to do nothing.

**modified**

Answer true if the receiver has been modified. Implemented by subclasses.

**name**

Answer the pane name. Pane's name is given by sending the #setName: message to the pane. For backward compatibility with the old event system, if no pane name has been provided, it defaults to the selector for the #getContents event. #name: should not be used, and it is provided for backward compatibility only.

**noDefaultStyle**

Tell the receiver not to use its `defaultStyle` method as the basis for subsequent style settings (start from scratch).

**noGroupLeader**

Make the receiver not be a group leader. By default every subpane is not a group leader.

**noTabStop**

Make the receiver not be a tabstop. By default every subpane which can be a tabstop is a tabstop. Send this message if you want to make it not be a tabstop.

**owner**

Answer the owner of the receiver.

**owner:** *anObject*

Set the owner of the receiver to *anObject*.

**paneName**

Answer the name of the pane.

**paneName:** *aString*

Set the name of the widget to *aString*. When in the context of a `ViewManager`, the widget may be retrieved with the following code: `self paneNamed: aString`.

**parent**

Answer the receiver's parent window.

**pen**

Answer the `graphicsTool` to be used for drawing.

**performWhenValid**

Perform the messages that require a valid handle (created by calls to `#whenValid:`)

**popup**

Answer the popup instance variable.

**propertyAt:** *key*

Answer the value associated with *key* in the properties dictionary.

**propertyAt: key ifAbsent:** *aBlock*

Answer the value associated with *key* in the properties dictionary; if absent, answer the result of evaluating *aBlock*.

**propertyAt: key ifAbsentPut: aBlock**

Answer the value associated with *key* in the properties dictionary; if absent, evaluate *aBlock*, put the evaluation result at *key*, and answer the result.

**propertyAt: key put: aValue**

Set the *value* associated with *key* in the properties dictionary.

**realInvalidateRect: aRectangle**

Invalidate *aRectangle* in the receiver. Causes the window to be repainted.

**receiveAllWindowsMessages**

Subclass the host control for the receiver so that all host messages will be received.

**rectangle**

Answer the bounding rectangle of the receiver.

**rectangle: aRectangle**

Set the position of the receiver to *aRectangle*.

**redraw**

Force the entire widget will be redrawn.

**removeBorderStyle**

Remove the border style from the receiver.

**removeHorizontalScrollbarStyle**

Remove the horizontal scrollbar style.

**removeTransparentStyle**

Remove the transparent style.

**removeVerticalScrollbarStyle**

Remove the vertical scrollbar style.

**resize: aRectangle**

Resize the receiver base on the parent's rectangle.

**scrollBoth**

Add horizontal and vertical scroll bars to the receiver.

**scrollBy: aPoint scrollRect: scrollRect clipRect: clipRect flags: anInteger**

Scroll the receiver by *aPoint* in the x and y directions. The flags are platform-dependent scrolling options.

**scrollHorizontally**

Add a horizontal scroll bar to the receiver.

**scrollVertically**

Add a vertical scroll bar to the receiver.

**selection**

Answer the receiver's current selection. Usually implemented by subclasses.

**selection:** *aSelection*

Set the receiver's current *selection*. Usually implemented by subclasses.

**setFocus**

Set the focus to the widget.

**setFocusOnControl**

Set the focus on receiver. This is different than 'setFocus' in that this also changes the border of the control.

**setMenu:** *aMenu*

Set the menu of the widget to *aMenu*.

**setName:** *aString*

Set the pane name to *aString* which can be a Symbol or a String.

**setPopupMenu:** *aMenu*

Set the popup menu of the widget to *aMenu*.

**setStyle:** *newStyle*

Set the receiver's host style to *newStyle*.

**showCaret:** *aBoolean*

Show or hide the caret.

**showHorizontalScrollBar:** *aBoolean*

Hide or show the horizontal scroll bar on the receiver.

**showVerticalScrollBar:** *aBoolean*

Hide or show the vertical scroll bar on the receiver.

**showWindow**

Make the widget visible.

**startGroup**

Make the control pane be the start of a group of dialog items. Arrow keys will cycle within a group of dialog items which starts from this pane up to (but not including) the next pane which this message has been sent to. This message has an effect only if the control pane is a part of a dialog window.

**superWindow**

Answer the non-frame parent window of the receiver.

**tabGroupMembers**

Answer the members of the tab group that the receiver belongs to.

**tabStop**

Make the dialog item receive the focus when the user presses the TAB key.

**tipText**

Answer the tip text for the receiver.

**tipText:** *aString*

Set the tip text for the receiver to *aString*.

**updateSelection**

The receiver's selection has changed, update it.

**visible**

Answer true if the window is visible

**when:** *anEvent* **perform:** *aSelector*

**(obsolete)**

Notify the owner of the widget whenever *anEvent* occurs by performing *aSelector*. *aSelector* takes one argument, the widget itself, and must be a method which the widget's owner can understand.

**when:** *eventName* **send:** *selector*

Form an action with the window's owner as the receiver and a *selector* as the message selector and append it to the actions list for the event named *eventName*.

**when:** *eventName* **send:** *selector* **to:** *anObject*

Form an action with *anObject* as the receiver and a *selector* as the message selector and append it to the actions list for the event named *eventName*.



**when:** *eventName* **send:** *selector* **to:** *anObject* **withArgument:** *argument*

Form an action with *anObject* as the receiver and a *selector* as the message selector and append it to the actions list for the event named *eventName*. Use *argument* as the argument to the last event parameter.

**whenValid**

Answer the list of messages that must be sent to the receiver's host window after the handle is valid (the window has been created).

**whenValid:** *selector*

Add a message to the list of messages that must be sent to the receiver's host window after the handle is valid (the window has been created).

**whenValid:** *selector* **with:** *anArgument*

Add a message to the list of messages that must be sent to the receiver's host window after the handle is valid (the window has been created).

**whenValid:** *selector* **withArguments:** *anArray*

Add a message to the list of messages that must be sent to the receiver's host window after the handle is valid (the window has been created).

**width**

Answer the receiver's width.

## Supported Events

**changed**

This event occurs when the contents of a widget have changed.

**backTabbed**

This event occurs when the user presses the TAB and SHIFT keys.

**display**

This event occurs when the widget wants to display itself. This event is only received by `GraphPane` and its subclasses.

**dragSourceCut:** *dragSession*

This event occurs when a drag-drop object for which the window was the source has been transferred and the source object now needs to be cut.

**dragSourceNeedsObject:** *dragSession*

This event occurs when the widget is a drag-drop source and the object is needed.

**dragTargetDrawEmphasis:** *dragSession*

This event occurs when emphasis is drawn indicating where the drop would occur in the widget.

**dragTargetDrop:** *dragSession*

This event occurs when a drag-drop object is dropped over a widget, if the widget is the target candidate.

**dragTargetEnter:** *dragSession*

This event occurs when the mouse enters the widget during a drag-drop session and the widget is the target candidate.

**dragTargetEraseEmphasis:** *dragSession*

This event occurs when the last drawn emphasis is removed.

**dragTargetLeave:** *dragSession*

This event occurs when the mouse leaves the window during a drag-drop session and the widget is a target candidate.

**dragTargetNeedsCursors:** *dragSession*

This event occurs when the widget needs the cursors during a drag-drop session.

**dragTargetNeedsOperations:** *dragSession*

This event occurs when the widget needs the supported operations during a drag-drop session.

**dragTargetOver:** *dragSession*

This event occurs when the mouse moves within the widget boundary during a drag-drop session.

**gettingFocus**

This event occurs when the widget is receiving input focus.

**help**

This event occurs whenever the F1 key is pressed when the widget has focus.

**losingFocus**

This event occurs when the widget is losing input focus.

**mouseMoved:** *aPoint*

This event occurs when the mouse moves over any widget.

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the `#contents:` method.

**needsMenu**

This event occurs whenever a widget looks for its regular menu (as opposed to its popup menu). This event is not used in conjunction with `WindowBuilder` - use **needsPopupMenu** instead.

**needsPopupMenu**

This event occurs whenever a widget looks for its popup menu (generally as a result of a right button click).

**needsSelection**

This event occurs when the user's selection has changed.

**resized**

This event occurs after any widget is resized.

**rightClicked**

This event occurs when the right mouse button is clicked in the receiver.

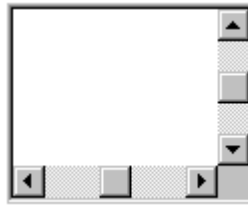
**tabbed**

This event occurs when the user presses the tab key.



# AnimationPane

*ParcPlace-Digitalk*



AnimationPanes provide a mechanism to perform simple animations. The widget contains a collection of *AnimatedObjects* which it manipulates. When the animation is started, a background process is started which sends messages to each active *AnimatedObject*. For complete details, see Digitalk's documentation or refer to Digitalk's Graphics Demo as an example of how to use them.

## Protocol

**addObject:** *anAnimatedObject*

Add *anAnimatedObject* to the widget's collection of animated objects.

**animate:** *anAnimatedObject*

Start *anAnimatedObject* moving continuously.

**clear**

Stop all animation and remove all objects.

**contents:** *aCollection*

Set the widget's collection of animated objects

**deanimate:** *anAnimatedObject*

Stop *anAnimatedObject* moving continuously.

**go**

Start animating all active objects.

**isActive:** *anAnimatedObject*

Answer true if *anAnimatedObject* is being animated.

**makeActive:** *anAnimatedObject*

Make anAnimatedObject active, but do not start it animating.

**stopAll**

Stop all objects from animating.

## Supported Events

**button1Down**

This event occurs whenever the left mouse button has been pressed down within the pane.

**button1DownShift**

This event occurs whenever the left mouse button and SHIFT key have been pressed down within the pane.

**button1Moved**

This event occurs whenever the left mouse button is down and the mouse is moved.

**button1UpShift**

This event occurs whenever the left mouse button is down and the SHIFT key is released.

**button2DoubleClicked**

This event occurs when the user double clicks with the right mouse button within the pane.

**button2Down**

This event occurs whenever the right mouse button has been pressed down within the pane.

**button2Moved**

This event occurs whenever the right mouse button is down and the mouse is moved.

**clicked:** *aPoint*

This event occurs whenever the left mouse button has been released.

**display**

This event occurs whenever the widget wants to display itself.

**doubleClicked**

This event occurs when the user double clicks with the left mouse button within the pane. Use the mouseLocation method to determine the position.

**mouseMoved:** *aPoint*

This event occurs whenever the mouse is moved within the pane.


**rightClicked**

This event occurs whenever the right mouse button has been released.



# Button

*ParcPlace-Digitalk*



Buttons provide a mechanism to initiate an action when clicked on. They may be labeled with a text string and will generate a clicked event when pressed. Buttons are commonly used to close the current window, open other windows, and perform actions on data associated with the current window.

## Protocol

### **cancelPushButton**

Set the cancel push button style so that this button is ‘clicked’ when the escape key is pressed.

### **click**

Programatically click the button.

### **contents**

Answer the label of the button.

### **contents:** *aString*

Set the label of the button to the string specified by *aString*.

### **defaultPushButton**

Set the button to be the default push button. Pressing the Enter key is equivalent to clicking this button.

### **label**

Answer the label of the button.

### **label:** *aString*

Set the label of the receiver to *aString*.

### **pushButton**

Set the Button to be a simple push button.

**setLabel:** *aString*

Set the label of the receiver to *aString*. Answer *aString*.

## Supported Events

**clicked**

This event occurs any time the button is pressed





# CheckBox

*ParcPlace-Digitalk*

- ☒ **Bold**
- ☐ **Italic**
- ☐ **Underline**

CheckBoxes are used to represent a boolean value that is either on or off (true or false). These buttons not only allow the user to set or change a boolean value, but they also act as an indicator of the value's current state.

## Protocol

### **autoCheckBox**

Set the automatic style. The widget will manage its own state and automatically toggle itself.

### **check**

Set the receiver's state to checked (true). Trigger changed event.

### **checkBox**

Set the simple style. The developer must manage the CheckBox's state when clicked on.

### **click**

Programatically click the button.

### **contents**

Answer the label of the button.

### **contents:** *aString*

Set the label of the button to the string specified by *aString*.

### **label**

Answer the label of the button.

### **label:** *aString*

Set the label of the receiver to *aString*.

**selection**

Answer the state of the CheckBox - true if it is on, false if off.

**selection:** *aBoolean*

Set the state of the CheckBox - true to check it, false to uncheck it.

**setLabel:** *aString*

Set the label of the receiver to aString. Answer *aString*.

**setValue:** *aBoolean*

Set the state of the receiver to *aBoolean*. Trigger changed event if the value has changed. Answer *aBoolean*.

**uncheck**

Set the receiver's state to unchecked (false). Trigger changed event.

**value**

Answer the value of the receiver as a boolean.

**value:** *aBoolean*

Set the state of the receiver to *aBoolean*.

## Supported Events

**clicked:** *aBoolean*

This event occurs any time the button is pressed.

**checked**

This event occurs any time the widget is turned on.

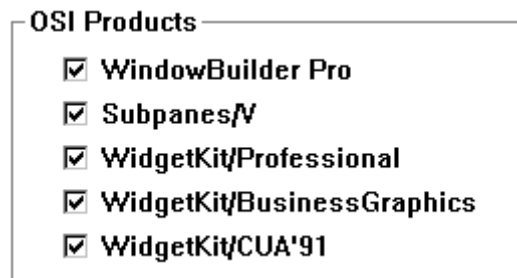
**unchecked**

This event occurs any time the widget is turned off.



# CheckBoxGroup

Objectshare



CheckBoxGroup is a special CompositePane subclass that provides a mechanism for the user to select multiple options from a set of options. CheckBoxGroups share the same protocol as MultipleSelectListBoxes and may be used interchangeably.

## Protocol

### contents

Answer the collection of labels of the CheckBoxes within the group.

**contents:** *aCollectionOfStrings*

Set the labels of the CheckBoxes within the group. The number of CheckBoxes is determined by the size of *aCollectionOfStrings*. **Note: This method may only be used before the window is opened.**

**indexOf:** *aString*

Answer the index of the item *aString*.

### label

Answer the label of the GroupBox surrounding the widget.

**label:** *aString*

Sets the text label of the GroupBox surrounding the widget.

### list

Return the collection of labels of the buttons within the group.

**list:** *aCollectionOfStrings*

Set the labels of the buttons within the group. The number of buttons is determined by the size of *aCollectionOfStrings*. **Note: This method may only be used before the window is opened.**

**numColumns**

Answer the number of columns in the group.

**numColumns:** *anInteger*

Set the number of columns in the group.

**selectedItems**

Answer a collection of the selected items.

**selectIndex:** *index*

Select the button at the specified *index*.

**selection**

Answer the index of the first item selected. The index starts at 1.

**selection:** *anObj*

If *anObj* is a collection then select items whose indices are in *anObj*. If *anObj* is an Integer then select the item indexed by *anObj*. Otherwise, select *anObj* in the list.

**selections**

Answer the indices of the selected items.

**unSelectIndex:** *index*

Unselect the button at the specified *index*.

## Supported Events

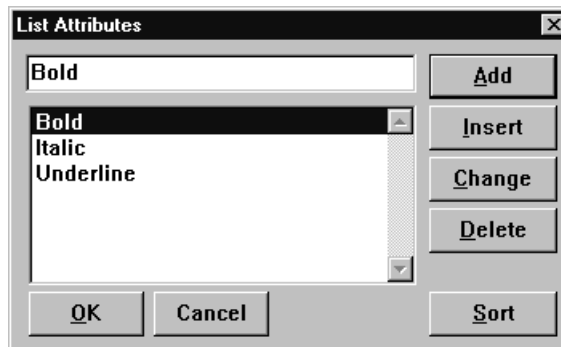
**clicked:** *aString*

This event occurs whenever one of the grouped CheckBoxes is clicked.

**changedIndex:** *anIndex*

This event occurs whenever one of the grouped CheckBoxes is clicked.

## WindowBuilder Extensions



CheckBoxGroups share the same attribute editor with ListBoxes, ComboBoxes, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



## ButtonListBox

*ParcPlace-Digitalk*



ButtonListBoxes provide the ability to toggle a list of choices on and off. They also provide single select listbox capabilities.

### Protocol

**checkIndex:** *anIndex*

Set the state of the item indexed by *anIndex* to checked, and redraw the item.

**checkItem:** *anItem*

Set the state of the item *anItem* to checked, and redraw the item.

**clearSelection**

Make no list items selected

**contents**

Answer the list.

**contents:** *aCollectionOfObjects*

Set the contents of the widget to *aCollectionOfObjects*.

**deleteAll**

Delete the entire list.

**deleteIndex:** *index*

Delete item number *index* from the list.

**deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**deselect**

Deselect the current selection.

**disableNoScroll**

Add the disable no scroll style.

**getTopIndex**

Answer the index of the first visible item.

**getImageFor:** *aButtonListBoxItem*

Answer the image for *aButtonListBoxItem*.

**getStateFor:** *aButtonListBoxItem*

Answer the state for *aButtonListBoxItem*.

**getStringFor:** *aButtonListBoxItem*

Answer the string for *aButtonListBoxItem*.

**indeterminateIndex:** *anIndex*

Set the state of the item indexed by *anIndex* to indeterminate, and redraw the item.

**indeterminateItem:** *anItem*

Set the state of the item *anItem* to indeterminate, and redraw the item.

**indexOf:** *aString*

Answer the index of *aString* in the list.

**insertItem:** *aString*

Insert the item *aString* into the list.

**insertItem:** *item* **at:** *index*

Insert the given *item* into the list (and the host control) at the given position.

**integralHeight**

Remove the no integral height style.

**itemHeight**

Answer the height of a list item.

**itemHeight:** *anInteger*

Set the height of a list item; only has an effect before the receiver window is opened.

**itemIndexFromPoint:** *aPoint*

Answer the index of the element under *aPoint* (where *aPoint* is relative to the receiver window).

**itemStates:** *aCollection*

Set the collection containing the states of the items to *aCollection*; each item's state may have the value true (checked), false (unchecked), or nil (indeterminate).

**lineAt:** *index*

Answer the line at index

**list**

Answer the list in the receiver.

**list:** *anArray*

Set the list in the receiver.

**restore**

Refresh the list from the owner and maintain the position in the list without selecting it.

**restoreSelected**

Refresh the list from the owner and keep the old selection.

**restoreWithRefresh:** *anObject*

Refresh the list from the owner and keep the line containing *anObject* visible and selected.

**selectedItem**

Answer the item currently selected in the widget.

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the index of the item currently selected in the widget.

**selection:** *anObject*

Select the line indicated by *anObject*. *anObject* may either be an index into the list or a string contained in the list.

**setColumnWidth:** *anInteger*

Set the column width for columns in a listbox with the multiColumn style to *anInteger* pixels



**setHorizontalExtent:** *pixelWidth*

Sets the width in pixels by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will scroll items in the list box. If the list box is as large larger than this value, the horizontal scroll bar is disabled.

**setList:** *aCollection*

Set the receiver's list to aCollection. Answer *aCollection*.

**setTabStops:** *anArray*

Set the tab stop positions in the receiver to *anArray*; the receiver must have been created with the useTabStops style; tab stop positions are specified in dialog units

**setTopIndex:** *anInteger*

Set the first visible item in the receiver to be the item at *anInteger*.

**setValue:** *anItem*

Set the selection in the receiver's list to *anItem*. Trigger changed event if the selection has changed. Answer *anItem*.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

**threeState:** *aBoolean*

Set whether the receiver is a three-state style button list box (default is false).

**uncheckIndex:** *anIndex*

Set the state of the item indexed by *anIndex* to unchecked, and redraw the item.

**uncheckItem:** *anItem*

Set the state of the item *anItem* to unchecked, and redraw the item.

**useImages:** *aBoolean*

Determinate whether any specified images in the list contents are to be used in displaying the list items (default is true if images have been supplied).

**useTabStops**

Add the use tab stops style.

**value**

Answer the selected item in the list.

**value:** *anItem*

Set the selection in the receiver's list to *anItem*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

**aboutToChange**

This event occurs whenever the selection is about to change.

**changed:** *selectedItem*

This event occurs whenever the selection is changed (passes the item as the argument).

**changedIndex:** *selectedIndex*

This event occurs whenever the selection is changed (passes the index as the argument).

**characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

**checked:** *aString*

This event occurs whenever any item is checked on.

**checkedIndex:** *anIndex*

This event occurs whenever any item is checked on.

**clicked:** *selectedItem*

This event occurs whenever the left mouse button is clicked.

**doubleClicked:** *selectedItem*

This event occurs whenever an item has been double clicked.

**indeterminate:** *aString*

This event occurs whenever any item is set to the indeterminate state.

**indeterminateIndex:** *anIndex*

This event occurs whenever any item is set to the indeterminate state.

**needsImageFor:** *anItem*

This event occurs whenever an item needs to display its image.

**needsStringFor:** *anItem*

This event occurs whenever an item needs to display its string.

**needsStateFor:** *anItem*

This event occurs whenever an item needs to display its state.

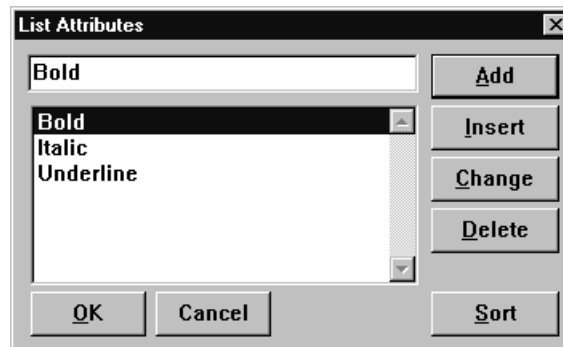
**unchecked:** *aString*

This event occurs whenever any item is checked off.

**uncheckedIndex:** *anIndex*

This event occurs whenever any item is checked off.

## WindowBuilder Extensions



ButtonListBoxes share the same attribute editor with ComboBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



# ComboBox

*ParcPlace-Digitaltalk*



ComboBoxes are hybrids between EntryFields and ListBoxes. It consists of a text entry field which is always visible and a list which may be displayed all of the time or only when the user selects the control.

ComboBoxes come in three varieties: simple, dropDown, and dropDownList. Simple ComboBoxes always display their lists. A dropDown ComboBox has a button next to the entry field which when clicked causes the list to appear. A dropDownList ComboBox only allows text entries that are items in the list.

## Protocol

### **clear**

Clear the contents of the receiver.

### **clearSelection**

Make no list items selected

### **clearTextSelection**

Clear the entry field selection from the control; can't use #clearSelection, because that is defined differently in ListBox

### **copySelection**

Copy the selection into the Clipboard

### **cutSelection**

Cut the selection into the Clipboard

### **contents**

Answer the contents of the ComboBox.

**contents:** *aCollection*

Set the contents of the ComboBox to *aCollection*.

**deleteAll**

Delete the whole list.

**deleteIndex:** *index*

Delete item number *index* from the list.

**deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**indexOf:** *aString*

Answer the index of the item *aString* in the list.

**insertItem:** *aString*

Insert *aString* into the list.

**deleteAll**

Delete the entire list

**dropDown**

Set the style of the ComboBox to **dropDown**.

**dropDownList**

Set the style of the ComboBox to **dropDownList**.

**entryField**

Answer the control window for the EntryField part of the receiver.

**indexOf:** *aString*

Answer the index of the item *aString*.

**insertItem:** *aString*

Append an item containing *aString* to the receiver's list.

**insertItem: item at: index**

Insert the given *item* into the list (and the host control) at the given position.

**list**

Answer the list in the receiver.

**list:** *anArray*

Set the list in the receiver.

**pasteSelection**

Paste the Clipboard contents into the control

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a `#changed:` handler) or the last time `previousValue` was sent.

**previousValue:** *aValue*

Set the previous value of the receiver.

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control; default = `#printString`.

**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if none is set, non-String objects are sent `#printString`.

**selectedItem**

Answer the item selected in the ListBox

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the index of the selected item. The index starts at 1.

**selection:** *anObject*

Select the item indicated by *anObject*. *anObject* is either an index into the list or a string with which to search the list.

**setList:** *aCollection*

Set the receiver's list to *aCollection*. Answer *aCollection*.

**setValue:** *aString*

Set the text of the entry field in the combo box to *aString*. Trigger the changed event if the value is different. Answer *aString*.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

**showDropdown:** *aBoolean*

Show or hide the listbox part of the receiver.

**simpleList**

Set the style of the ComboBox to simple.

**text**

Answer the text in the ComboBox's text entry field.

**text:** *aString*

Set the text of the ComboBox to aString.

**value**

Answer the value in the entry field of the combo box.

**value:** *aString*

Set the text of the entry field in the combo box to *aString*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

**aboutToChange**

This event occurs whenever the selection is about to change.

**changed:** *aString*

This event occurs whenever the selection is changed.

**characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

**clicked:** *aString*

This event occurs whenever the left mouse button is clicked.

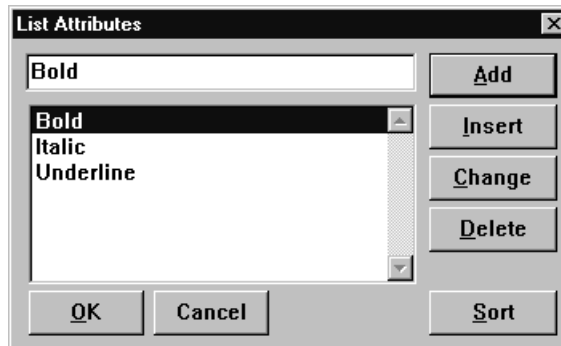
**doubleClicked:** *aString*

This event occurs whenever an item has been double clicked.

**textChanged:** *aString*

This event occurs any time the text of the entry field changes. This event is sent when the contents of the ComboBox is set, the user types into the entry field, or the user selects from the list.

## WindowBuilder Extensions



ComboBoxes share the same attribute editor with ListBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.





## DrawnButton

*ParcPlace-Digitalk*



DrawnButtons act exactly like regular push buttons. They are labeled with graphic images rather than text strings. We have also provided an invisible style which can be used to create “sense regions” on other graphic images. For example, you could place invisible buttons on top of a large imported bitmap.

### Protocol

#### **click**

Programatically click the button.

#### **contents**

Answer the bitmap displayed by the button

#### **contents:** *aBitmap*

Set the contents of the button to the bitmap specified by *aBitmap*.

#### **fixedSize**

If the widget has a bitmap as its label, draw the bitmap at its normal size.

#### **invisible**

Makes the button invisible (no label or bitmap). This is useful for creating “sense regions” on top of other visual objects. For this to work properly, exclude the widget from the tab order.

#### **label**

Answer the label of the button.

#### **label:** *aBitmap*

Set the label of the receiver to *aBitmap*.

#### **setLabel:** *aBitmap*

Set the label of the receiver to *aBitmap*. Answer *aBitmap*.

**stretchToFit**

If the widget has a bitmap as its label, stretch or shrink the bitmap to fill the widget.

## Supported Events

**clicked**

This event occurs any time the button is pressed

**drawItem**

This event occurs whenever the item needs to be drawn

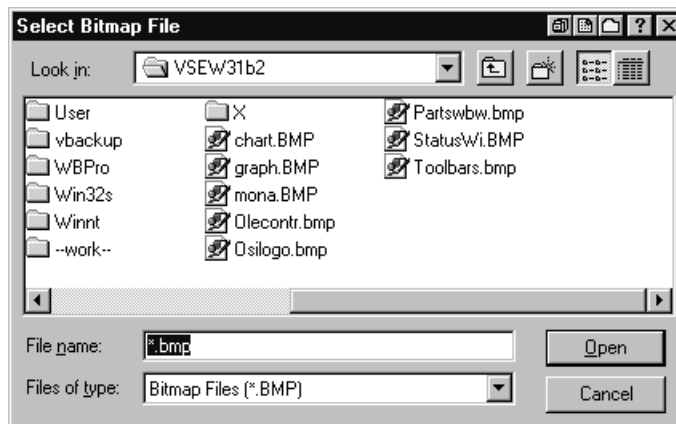
**drawFocus**

This event occurs whenever the item needs to be drawn with focus

**drawSelection**

This event occurs whenever the item needs to be drawn selected

## WindowBuilder Extensions



The attribute editor for DrawnButtons allows you to select a bitmap file (.BMP) from disk.



## DropDownList

*ParcPlace-Digitalk*



DropDownLists are hybrids between EntryFields and ListBoxes. It consists of a read-only text entry field which is always visible and a list which is displayed whenever the user clicks on the control. A widget only allows text entries that are items in the list.

### Protocol

#### **clear**

Clear the contents of the receiver.

#### **clearSelection**

Make no list items selected

#### **clearTextSelection**

Clear the entry field selection from the control; can't use #clearSelection, because that is defined differently in ListBox

#### **copySelection**

Copy the selection into the Clipboard

#### **cutSelection**

Cut the selection into the Clipboard

#### **contents**

Answer the contents of the ComboBox.

#### **contents:** *aCollection*

Set the contents of the ComboBox to *aCollection*.

#### **deleteAll**

Delete the whole list.

**deleteIndex:** *index*

Delete item number *index* from the list.

**deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**indexOf:** *aString*

Answer the index of the item *aString* in the list.

**insertItem:** *aString*

Insert *aString* into the list.

**deleteAll**

Delete the entire list

**indexOf:** *aString*

Answer the index of the item *aString*.

**insertItem:** *aString*

Append an item containing *aString* to the receiver's list.

**insertItem:** *item at: index*

Insert the given *item* into the list (and the host control) at the given position.

**list**

Answer the list in the receiver.

**list:** *anArray*

Set the list in the receiver.

**pasteSelection**

Paste the Clipboard contents into the control

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control;  
default = #printString.

**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if  
none is set, non-String objects are sent #printString.

**selectedItem**

Answer the item selected in the ListBox

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the index of the selected item. The index starts at 1.

**selection:** *anObject*

Select the item indicated by *anObject*. *anObject* is either an index into the list or a string with which to search the list.

**setList:** *aCollection*

Set the receiver's list to *aCollection*. Answer *aCollection*.

**setValue:** *aString*

Set the text of the entry field in the combo box to *aString*. Trigger the changed event if the value is different. Answer *aString*.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

**showDropdown:** *aBoolean*

Show or hide the listbox part of the receiver.

**value**

Answer the value in the entry field of the combo box.

**value:** *aString*

Set the text of the entry field in the combo box to *aString*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

**aboutToChange**

This event occurs whenever the selection is about to change.

**changed:** *aString*

This event occurs whenever the selection is changed.

**characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

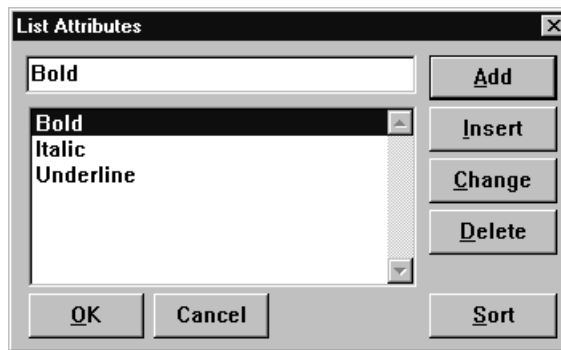
**clicked:** *aString*

This event occurs whenever the left mouse button is clicked.

**listVisible**

This happens when the user clicks the DropDownList button to pull down the list.

## WindowBuilder Extensions



DropDownLists share the same attribute editor with ListBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



## EnhancedEntryField

*Objectshare*



EnhancedEntryFields 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 **readonly** and **password** are available. The maximum number of allowable characters may be set. Justification (right, left, centered) may be specified

Under OS/2, auto-tabbing may be specified.

### Protocol

#### **autoHScroll**

Add the auto horizontal scroll style.

#### **autoTabStyle**

Set the entry field to automatically tab to the next field when the maximum number of characters has been entered. Currently not supported under Windows.

#### **autoVScroll**

Add the auto vertical scroll style.

#### **case**

Answer the receiver's case (UPPER/lower).

#### **case:** *aSymbol*

Specify any automatic case conversion that should take place when the field loses focus. Allowable values are: **#UPPER**, **#lower**, **#Proper**, and **#Unchanged**.

**centered**

Specify center justification. Currently not supported under Windows.

**character**

Answer the character validation selector.

**character:** *aSymbol*

Set the character level validation function.

**clear**

Clear the contents of the receiver.

**clearSelection**

Delete the current selection.

**contents**

Answer the contents of the field as a string.

**contents:** *aString*

Set the contents of the field adjusting the case as required.

**copySelection**

Copy current selection to the clipboard.

**cutSelection**

Copy current selection to the clipboard then delete it from the text.

**field**

Answer the field validation selector.

**field:** *aSymbol*

Set the field level validation function.

**getFocus**

Answer where the cursor should be placed when the field gets focus.

**getFocus:** *aSymbol*

Specify where the cursor should be placed when the field gets focus. Allowable values include: **#selectAll**, **#selectFirst**, and **#selectLast**.

**getSelection**

Answers the starting and ending character positions of the current selection of the field.



**insertSelectedText:** *aString* **at:** *anInteger*

Insert *aString* into the receiver immediately before index *anInteger* and have the newly inserted text selected.

**left**

Specify left justification. This is the default.

**lowerCase**

Add the lower case style.

**modified:** *aBoolean*

Set whether the field has been modified.

**nextPut:** *aCharacter*

Add a character at the end of the text in the pane.

**nextPutAll:** *aString*

Add *aString* at the end of the text in the pane.

**password**

Make the field password protected.

**pasteSelection**

Replace selected text with the clipboard contents.

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a `#changed:` handler) or the last time `previousValue` was sent.

**previousValue:** *aValue*

Set the previous value of the receiver. Adjust the case first.

**readOnly**

Make the field read only.

**readOnly:** *aBoolean*

Set the readonly property of the receiver to *aBoolean*.

**readWrite**

Clear the readonly property of the receiver.

**retryChange**

Stop the change process, and set the focus back to allow the user to modify the receiver and try again.

**right**

Specify right justification. Currently not supported under Windows.

**selectAll**

Select all of the text in the field.

**selectFirst**

Place cursor at beginning of the text.

**selectFrom:** *aStartIndex* **to:** *anEndIndex*

Select the specified range of text.

**selectLast**

Place cursor at end of the text.

**setTextLimit:** *anInteger*

Set the maximum number of characters that the receiver can hold to *anInteger*.

**setValue:** *aValue*

Set the text contents of the receiver to the string representing *aValue*. First set the value, with a possibility that it may be undone. Then, trigger `aboutToChangeTo:`, where the value may be restored to the previous value, or it may be set to an entirely new value. At the end, if the final value is different from the beginning value, trigger the changed event if the value is changed. Answer the final value.

**upperCase**

Add the upper case style.

**value**

Answer the text contents of the receiver.

**value:** *aString*

Set the text contents of the receiver to *aString*.

**wantReturn**

Add the want return style.

**ok...**

These are the field level validation methods. In addition to validating the field, they may also reformat the field. If the field is invalid, an error message will be presented to the user.

**ok...: aChar**

These are the character level validation methods. If the character is not valid, it will not be entered into the field.

## Supported Events

**aboutToChangeTo: aString**

This event occurs when the widget's value is about to change.

**changed: aString**

This event occurs when after the widget's value has changed.

**entered: aString**

This event occurs when the user presses the Enter key when this widget has focus.

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the `#contents:` method.

**objectChanged: anObject**

This event occurs when after the widget's value has changed. `anObject` will be an instance of a data type appropriate to the validation functions defined for the field.

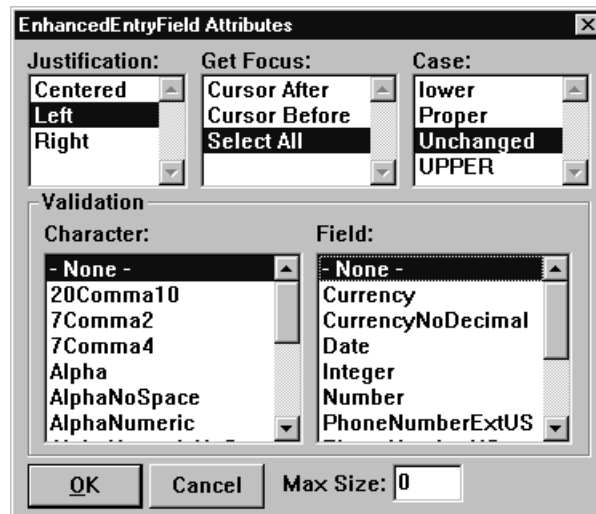
**objectChanging: anObject**

This event occurs whenever a character is typed within the field. `anObject` will be an instance of a data type appropriate to the validation functions defined for the field.

**textChanged: aString**

This event occurs whenever a character is typed within the field.

## WindowBuilder Extensions



The EnhancedEntryField attribute editor allows you to set the justification, get focus cursor position, case adjustment on losing focus, maximum number of characters, and auto-tabbing (OS/2 only).

Character and Field level validation may also be specified. The character level validations provided are:

### 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 .

The field level validations provided are:

**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.

**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.

Additional validation functions may be added easily. All of the validation functions are public methods of the EnhancedEntryField 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 EnhancedEntryField class.



## EntryField

*ParcPlace-Digitalk*



EntryFields are used to provide single line text entry and edit capabilities. If the user types more text than can be accommodated within the field, it will automatically scroll. Additional styles such as **readonly** and **password** are available. The **readonly** style places the field in output-only mode. Users can select and copy text within it but they can not change the text. The **password** style causes any entered values to be replaced by a series of asterisks (“\*”). The contents of the field may be pre-set by entering text into WindowBuilder’s text/label field.

### Protocol

#### **autoHScroll**

Add the auto horizontal scroll style.

#### **autoVScroll**

Add the auto vertical scroll style.

#### **clear**

Clear the contents of the receiver.

#### **contents**

Answer the contents of the field as a string.

#### **contents:** *aString*

Set the contents of the field to the string *aString*.

#### **clearSelection**

Delete the current selection.

#### **copySelection**

Copy current selection to the clipboard.

**cutSelection**

Copy current selection to the clipboard then delete it from the text.

**insertSelectedText:** *aString* **at:** *anInteger*

Insert *aString* into the receiver immediately before index *anInteger* and have the newly inserted text selected.

**lowerCase**

Add the lower case style.

**modified:** *aBoolean*

Set whether the field has been modified.

**nextPut:** *aCharacter*

Add a character at the end of the text in the pane.

**nextPutAll:** *aString*

Add *aString* at the end of the text in the pane.

**noHideSelection**

Add the no hide selection style.

**oemConvert**

Add the OEM convert style.

**password**

Make the field password protected.

**pasteSelection**

Replace selected text with the clipboard contents.

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a #changed: handler) or the last time previousValue was sent.

**previousValue:** *aValue*

Set the previous value of the receiver.

**readOnly**

Make the field read only.



**readWrite**

Clear the readonly property of the receiver.

**retryChange**

Stop the change process, and set the focus back to allow the user to modify the receiver and try again.

**selectAll**

Select all of the text in the field.

**selectFrom:** *aStartIndex* **to:** *anEndIndex*

Select the specified range of text.

**setTextLimit:** *anInteger*

Set the maximum number of characters that the receiver can hold to anInteger.

**setValue:** *aValue*

Set the text contents of the receiver to the string representing aValue. First set the value, with a possibility that it may be undone. Then, trigger aboutToChangeTo:, where the value may be restored to the previous value, or it may be set to an entirely new value. At the end, if the final value is different from the beginning value, trigger the changed event if the value is changed. Answer the final value.

**upperCase**

Add the upper case style.

**value**

Answer the text contents of the receiver.

**value:** *aString*

Set the text contents of the receiver to aString.

**wantReturn**

Add the want return style.

## Supported Events

**aboutToChangeTo:** *aString*

This event occurs when the widget's value is about to change.

**changed:** *aString*

This event occurs when after the widget's value has changed.

**entered:** *aString*

This event occurs when the user presses the Enter key when this widget has focus.

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the `#contents:` method.

**textChanged:** *aString*

This event occurs whenever a character is typed within the field.



## EntryFieldGroup

Objectshare

EntryFieldGroup is a special CompositePane subclass that provides a mechanism to rapidly create data entry forms. The widget may be displayed with or without a GroupBox or may optionally have a vertical scroll bar.

### Protocol

#### **contents**

Answers a dictionary that represents the contents of the widget. The keys of the dictionary are the StaticText labels. The value are the contents of the corresponding EntryFields.

**contents:** *aDictionaryOfStrings*

The dictionary's keys set the labels of the StaticText labels within the group. The dictionary's values provide the contents of the EntryFields. If *aDictionaryOfStrings* is some other type of collection than a dictionary, the items of the collection will be used as the StaticText labels and the EntryFields will be initialized to empty strings. The number of EntryFields is determined by the size of *aDictionaryOfStrings*. **Note: This method may only be used before the window is opened.**

#### **fieldClass**

Answers the EntryField subclass to be used within the widget. EntryFieldGroup can easily be subclassed to use a different text entry class (such as the EnhancedEntryField).

**label:** *aString*

Sets the text label of the GroupBox surrounding the widget.

#### **noGroupBox**

Sets the style of the widget to not include a surrounding GroupBox.

**setLabelFont:** *aFont*

Sets the font of the StaticText labels to *aFont*.

**setValueFont:** *aFont*

Sets the font of the EntryFields to *aFont*.

**verticalScrollBar**

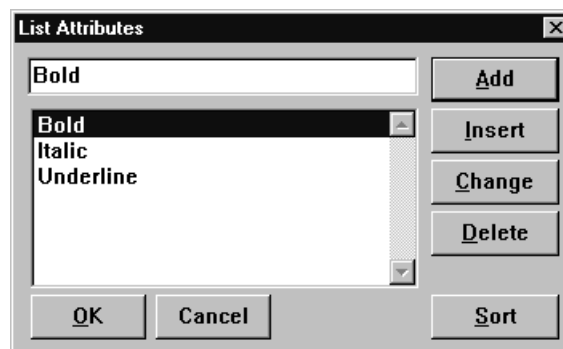
Sets the style of the widget to have a vertical scroll bar.

## Supported Events

**textChanged:** *aString* **field:** *aWidget*

This event occurs whenever the text of any one of the EntryFields is changed

## WindowBuilder Extensions

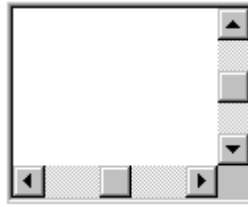


EntryFieldGroups share the same attribute editor with ListBoxes, ComboBoxes, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



## GraphPane

*ParcPlace-Digitalk*



GraphPanes are very powerful widgets that provide extensive support for visual display and user input. Using a GraphPane's pen is the easiest way to add custom graphics to an application. By associating a message with any of its mouse or keyboard events, a programmer can handle simple user interaction. For complete details, see Digitalk's documentation.

### Protocol

#### **bitmap**

Answer the bitmap associated with the receiver.

#### **bitmap:** *aBitmap*

Set the bitmap associated with the receiver.

#### **erase**

Clear the contents of the pane.

#### **mouseLocation**

Answer the mouse location as of the last mouse input event.

#### **noBorders**

Set the style to not include a border.

#### **noScrollBars**

Set the style not to include any scroll bars.

#### **stretch:** *anInteger*

Set the scaling attributes of the pane. 0 means no stretch. 1 means stretch while maintaining the aspect ration. Anything else means stretch according to the window's dimensions.

## Supported Events

### **button1Down**

This event occurs whenever the left mouse button has been pressed down within the pane.

### **button1DownShift**

This event occurs whenever the left mouse button and SHIFT key have been pressed down within the pane.

### **button1Moved**

This event occurs whenever the left mouse button is down and the mouse is moved.

### **button1UpShift**

This event occurs whenever the left mouse button is down and the SHIFT key is released.

### **button2DoubleClicked**

This event occurs when the user double clicks with the right mouse button within the pane.

### **button2Down**

This event occurs whenever the right mouse button has been pressed down within the pane.

### **button2Moved**

This event occurs whenever the right mouse button is down and the mouse is moved.

### **clicked:** *aPoint*

This event occurs whenever the left mouse button has been released.

### **display**

This event occurs whenever the widget wants to display itself.

### **doubleClicked**

This event occurs when the user double clicks with the left mouse button within the pane. Use the `mouseLocation` method to determine the position.

### **mouseMoved:** *aPoint*

This event occurs whenever the mouse is moved within the pane.

### **rightClicked**

This event occurs whenever the right mouse button has been released.



## GroupBox

*ParcPlace-Digitalk*

### GUI Tool

- ☐ WindowBuilder Pro
- ☐ PARTS Workbench
- ☐ VisualAge

GroupBoxes are used to visually indicate and label groups of related controls. They are comprised of a box with a label in the upper left corner.

## Protocol

### **label**

Answer the receiver's label

### **label:** *aString*

Set the receiver's label

### **setLabel:** *aString*

Set the receiver's label to *aString*. Answer *aString*.

## Supported Events

**None**



# Header

*ParcPlace-Digitalk - Windows 95 only*



The Header widget is a simple column header control. Each header item may specify its label, image, width and alignment. Events are generated whenever an item is clicked on or resized (via dragging the vertical line between items).

## Protocol

**buttons:** *aBoolean*

Determine whether the receiver has the button style, according to *aBoolean*.

**columnWidths**

Answer the current widths of heading columns (in pixels).

**contents**

Answer the contents of the receiver.

**contents:** *aCollection*

Set the headings to *aCollection*.

**deleteAll**

delete first item until no more items

**getAlignmentFor:** *aHeaderItem*

Answer the alignment for *aHeaderItem*.

**getImageFor:** *aHeaderItem*

Answer the image for *aHeaderItem*.

**getStringFor:** *aHeaderItem*

Answer the label for *aHeaderItem*.

**getWidthFor:** *aHeaderItem*

Answer the width for *aHeaderItem*.



**headings**

Answer the receiver's collection of headings (HeaderItems).

**headings:** *aCollection*

Set the headings to *aCollection*, where each element in *aCollection* is either a String or a HeaderItem.

## Supported Events

**clicked:** *aHeaderItem*

This event occurs whenever a item clicked.

**clickedIndex:** *index*

This event occurs whenever a item clicked.

**dividerDoubleClicked:** *aHeaderItem*

This event occurs whenever a divider is double clicked.

**dividerDoubleClickedIndex:** *index*

This event occurs whenever a divider is double clicked.

**needsAlignmentFor:** *aHeaderItem*

This event occurs whenever a header item's alignment is required.

**needsImageFor:** *aHeaderItem*

This event occurs whenever a header item's image is required.

**needsStringFor:** *aHeaderItem*

This event occurs whenever a header item's label is required.

**needsWidthFor:** *aHeaderItem*

This event occurs whenever a header item's width is required.

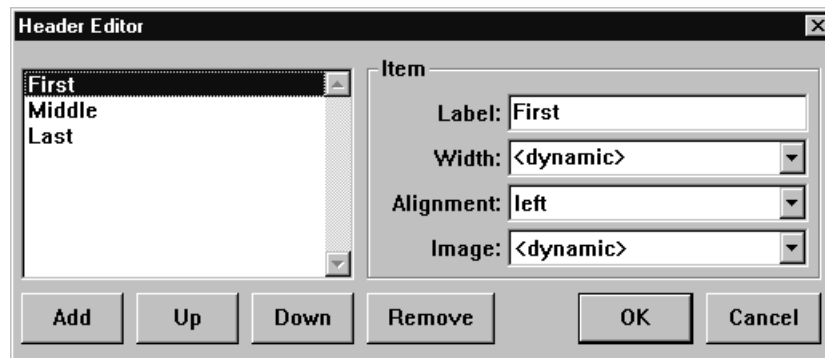
**resizedIndex:** *index to: newWidth*

This event occurs whenever a header item is resized.

**resizingIndex:** *index to: newWidth*

This event occurs whenever a header item is resizing.

## WindowBuilder Extensions



The attribute editor for Header widgets allows you to set up each individual header item. The **Add** button is used to add new items. The **Remove** button is used to remove items. The **Up** and **Down** buttons are used to re-order the items. For each item, you can specify its label, width, alignment and image. Width is expressed in pixels and may be specified as dynamic. Alignment can be left, right, center or dynamic. The image can be selected from the Bitmap Manager or specified as dynamic. Any option that is dynamic will be determined at runtime via firing an appropriate event (e.g., `#needsImageFor`: for specifying the image).



## ListBox

*ParcPlace-Digitalk*



ListBoxes provide a general selection capability. The user is presented with a list of choices (either strings or bitmaps) and may select one with the mouse.

### Protocol

#### **bitmaps**

Answer the list of bitmaps for owner drawing.

#### **bitmaps:** *aCollection*

Set the list of bitmaps for owner drawing to *aCollection*.

#### **clearSelection**

Make no list items selected

#### **contents**

Answer the list.

#### **contents:** *aCollection*

Set the contents of the widget to *aCollection*.

#### **deleteAll**

Delete the entire list.

#### **deleteIndex:** *index*

Delete item number *index* from the list.

#### **deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**deselect**

Deselect the current selection.

**disableNoScroll**

Add the disable no scroll style.

**drawBitmap: *aBitmap* for: *aString***

Use *aBitmap* in place of *aString* when displaying the contents of the list.

**getTopIndex**

Answer the index of the first visible item.

**indexOf: *aString***

Answer the index of *aString* in the list.

**insertItem: *aString***

Insert the item *aString* into the list.

**insertItem: *item* at: *index***

Insert the given *item* into the list (and the host control) at the given position.

**integralHeight**

Remove the no integral height style.

**itemHeight**

Answer the height of a list item.

**itemHeight: *anInteger***

Set the height of a list item; only has an effect before the receiver window is opened.

**itemIndexFromPoint: *aPoint***

Answer the index of the element under *aPoint* (where *aPoint* is relative to the receiver window).

**lineAt: *index***

Answer the line at index

**list**

Answer the list in the receiver.

**list: *anArray***

Set the list in the receiver.

**multiColumn**

Add the multi-column style.

**noIntegralHeight**

Add the no integral height style.

**noRedraw**

Add the no redraw style.

**notify**

Add the notify style.

**ownerDrawFixed**

Set the style of the widget to the fixed height owner draw style.

**ownerDrawVariable**

Set the style of the widget to the variable height owner draw style.

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control; default = #printString.

**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if none is set, non-String objects are sent #printString.

**restore**

Refresh the list from the owner and maintain the position in the list without selecting it.

**restoreSelected**

Refresh the list from the owner and keep the old selection.

**restoreWithRefresh:** *anObject*

Refresh the list from the owner and keep the line containing *anObject* visible and selected.

**selectedItem**

Answer the item currently selected in the widget.

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

### **selection**

Answer the index of the item currently selected in the widget.

### **selection:** *anObject*

Select the line indicated by *anObject*. *anObject* may either be an index into the list or a string contained in the list.

### **setColumnWidth:** *anInteger*

Set the column width for columns in a listbox with the multiColumn style to *anInteger* pixels

### **setHorizontalExtent:** *pixelWidth*

Sets the width in pixels by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will scroll items in the list box. If the list box is as large larger than this value, the horizontal scroll bar is disabled.

### **setList:** *aCollection*

Set the receiver's list to aCollection. Answer *aCollection*.

### **setTabStops:** *anArray*

Set the tab stop positions in the receiver to *anArray*; the receiver must have been created with the useTabStops style; tab stop positions are specified in dialog units

### **setTopIndex:** *anInteger*

Set the first visible item in the receiver to be the item at *anInteger*.

### **setValue:** *anItem*

Set the selection in the receiver's list to *anItem*. Trigger changed event if the selection has changed. Answer *anItem*.

### **setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

### **sort**

Add the sort style.

### **standard**

Add the standard style.

### **useTabStops**

Add the use tab stops style.

**value**

Answer the selected item in the list.

**value:** *anItem*

Set the selection in the receiver's list to *anItem*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

**aboutToChange**

This event occurs whenever the selection is about to change.

**changed:** *selectedItem*

This event occurs whenever the selection is changed (passes the item as the argument).

**changedIndex:** *selectedIndex*

This event occurs whenever the selection is changed (passes the index as the argument).

**characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

**clicked:** *selectedItem*

This event occurs whenever the left mouse button is clicked.

**doubleClicked:** *selectedItem*

This event occurs whenever an item has been double clicked.

**drawFocus:** *drawIndex*

This event occurs whenever the widget focus changes. The widget is rendered to indicate that it has input focus.

**drawItem:** *drawIndex*

This event occurs whenever the application receives a request to draw the widget.

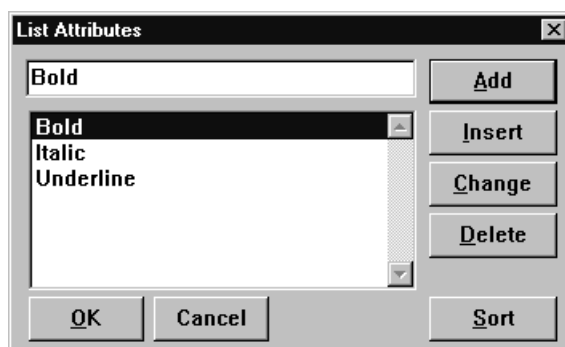
**drawSelection:** *drawIndex*

This event occurs whenever the widget receives input focus. The item is rendered to indicate that it has been selected.

**measureItem:** *drawIndex*

This event occurs whenever the height of an owner drawn item is set.

## WindowBuilder Extensions



ListBoxes share the same attribute editor with ComboBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.





## ListPane

*ParcPlace-Digital*



ListPanes provide a general selection capability and are functionally equivalent to ListBoxes. The user is presented with a list of choices and may select one with the mouse. In some VisualSmalltalk. platforms, ListPanes are implemented entirely in Smalltalk as opposed to ListBoxes which are real operating system widgets.

### Protocol

#### **characterTyped**

Answer the character typed. Meaningful when the #characterInput event is signaled.

#### **clearSelection**

Make no list items selected

#### **contents**

Answer the list.

#### **contents:** *aCollection*

Set the contents of the widget to *aCollection*.

#### **deleteAll**

Delete the entire list.

#### **deleteIndex:** *index*

Delete item number *index* from the list.

#### **deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**deselect**

Deselect the current selection.

**indexOf:** *aString*

Answer the index of *aString* in the list.

**insertItem:** *aString*

Insert the item *aString* into the list.

**insertItem: item at: index**

Insert the given item into the list (and the host control) at the given position.

**itemIndexFromPoint:** *aPoint*

Answer the index of the element under *aPoint* (where *aPoint* is relative to the receiver window).

**lineAt:** *index*

Answer the line at *index*.

**list**

Answer the list in the receiver.

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control; default = #printString.

**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if none is set, non-String objects are sent #printString.

**restore**

Refresh the list from the owner and maintain the position in the list without selecting it.

**restoreSelected**

Refresh the list from the owner and keep the old selection.

**restoreWithRefresh:** *aString*

Refresh the list from the owner and keep the line equal to *aString* showing and selected.

**selectedItem**

Answer the item currently selected in the widget.

**selectIndex:** *index*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the index of the item currently selected in the widget.

**selection:** *anObject*

Select the line indicated by *anObject*. *anObject* may either be an index into the list or a string contained in the list.

**selectItem:** *anObject*

Display the list with the line indicated by *anObject* selected. *anObject* is either the index into the list or a string with which the list is to be searched with.

**setList:** *aCollection*

Set the receiver's list to *aCollection*. Answer *aCollection*.

**setValue:** *anItem*

Set the selection in the receiver's list to *anItem*. Trigger changed event if the selection has changed. Answer *anItem*.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

**showSelection**

Highlight the selected line.

**topCorner:** *aPoint*

Change *topCorner* to *aPoint*.

**value**

Answer the selected item in the list.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

### **aboutToChange**

This event occurs whenever the selection is about to change.

### **changed:** *selectedItem*

This event occurs whenever the selection is changed (passes the item as the argument).

### **changedIndex:** *selectedIndex*

This event occurs whenever the selection is changed (passes the index as the argument).

### **characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

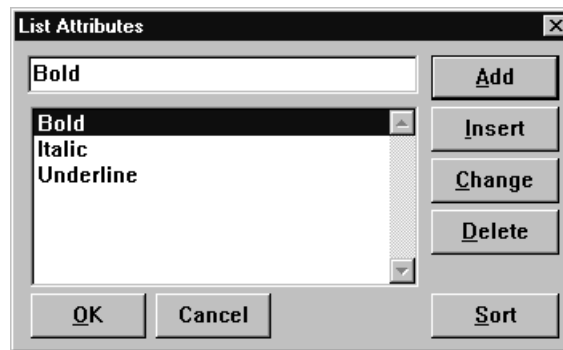
### **clicked:** *selectedItem*

This event occurs whenever the left mouse button is clicked.

### **doubleClicked:** *selectedItem*

This event occurs whenever an item has been double clicked.

## WindowBuilder Extensions

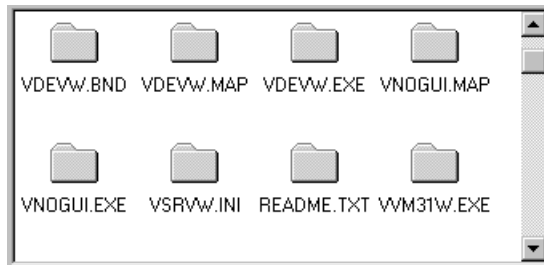


ListPanes share the same attribute editor with ComboBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



## Listview

*ParcPlace-Digitaltalk - Windows 95 only*



Name	Size	Type
VNOGUI.EXE	23552	Application
VSRVW.INI	12805	Configuration Settings
README.TXT	17824	Text Document
VVM31W.EXE	185856	Application
SUPPORT.TXT	7504	Text Document
PARTSwBw.BND	186	Smalltalk Bind File
PARTS.INI	616	Configuration Settings

Listview is an iconic container widget with multi-column list capabilities. The widget can display a collection of icons in various orientations. It can also supports a report view which acts like a multi-column listbox. Each column may specify a label, width and alignment.

### Protocol

#### **alignLeft**

Set and apply the align left style.

#### **alignTop**

Set and apply the align top style.

#### **autoArrange:** *aBoolean*

Enable or disable auto arrange, according to the value of *aBoolean*.

#### **contents**

Answer the contents of the receiver.

**contents:** *aCollection*

Set the contents of the receiver to *aCollection*.

**deleteAll**

Delete all items from the receiver.

**deleteAllColumns**

Delete all columns (headings) from the receiver.

**editLabels:** *aBoolean*

Enable or disable label editing, according to the value of *aBoolean*.

**getColumnsFor:** *aListItem*

Get the column values for *aListItem*.

**getImageFor:** *aListItem*

Get the image for *aListItem*.

**getSmallImageFor:** *aListItem*

Get the small image for *aListItem*.

**getStringFor:** *aListItem*

Get the label string for *aListItem*.

**headings**

Answer the receiver's collection of headings (HeaderItems).

**headings:** *aCollection*

Set the headings to aCollection.

**iconView**

Set and apply the (Normal) Icon View style.

**items**

Answer the receiver's collection of items.

**listView**

Set and apply the List View style.

**reportView**

Set and apply the Details (Report) View style.

**selectedIndex**

Answer the selected index.

**selectedItem**

Answer the selected item.

**selectIndex:** *anInteger*

Set the selected index to *anInteger*.

**selection**

Answer the selection.

**showSelectionAlways:** *aBoolean*

Determine whether the receiver displays its selection even when not active, according to the value of *aBoolean*.

**smallIconView**

Set and apply the Small Icon View style.

**sortAscending**

Answer whether the receiver is supposed to sort ascending or descending.

**sortAscending:** *aBoolean*

Set whether the receiver is supposed to sort ascending (true) or descending (false).

**sortByColumn:** *index*

Set the index of the column which is used to sort by to *index*.

**sortItems**

Sort the items using the current sort column and direction.

**useImages**

Answer whether any specified images in the list contents are to be used in displaying the list items (default is true if images have been supplied).

**useImages:** *aBoolean*

Determinate whether any specified images in the list contents are to be used in displaying the list items (default is true if images have been supplied).

## Supported Events

**aboutToChangeLabel:** *anItem to: newLabel*

This event occurs whenever an item's label is about to change.

**aboutToEditLabel:** *anItem*

This event occurs whenever an item's label is about to be edited.

**changed:** *anItem*

This event occurs whenever the selection is changed.

**changedLabel:** *anItem to: newLabel*

This event occurs whenever an item's label is changed.

**characterTyped:** *aCharacter*

This event occurs whenever a character is typed.

**clicked:** *anItem*

This event occurs whenever an item is clicked.

**clickedIndex:** *index*

This event occurs whenever an item is clicked.

**columnClicked:** *columnIndex*

This event occurs whenever a column is clicked.

**doubleClicked:** *anItem*

This event occurs whenever an item is double clicked.

**doubleClickedIndex:** *index*

This event occurs whenever an item is double clicked.

**needsColumnsFor:** *anItem*

This event occurs whenever an item's column values are required.

**needsImageFor:** *anItem*

This event occurs whenever an item's image is required.

**needsSmallImageFor:** *anItem*

This event occurs whenever an item's small image is required.

**needsStringFor:** *anItem*

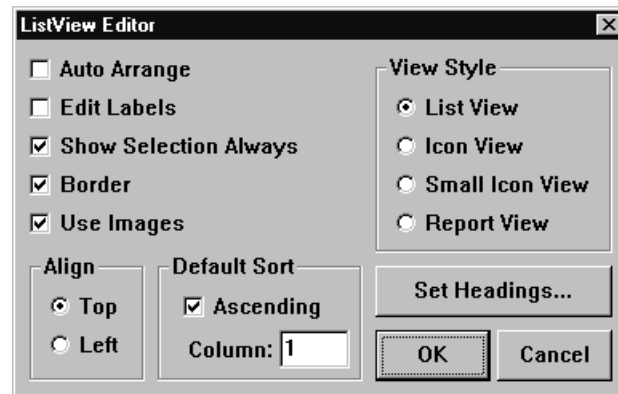
This event occurs whenever an item's label is required.



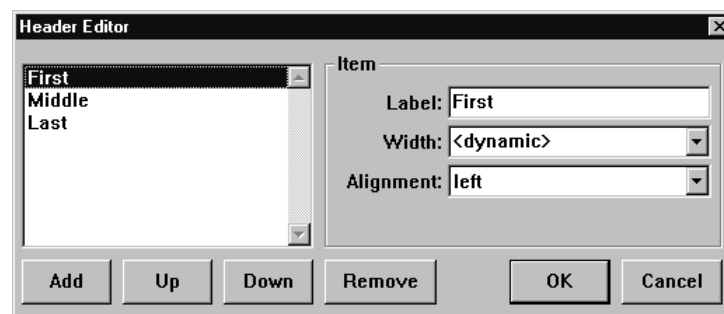
**selected:** *anItem*

This event occurs whenever an item is selected.

## WindowBuilder Extensions



The ListView editor allows you to set the view style to List View, Icon View, Small Icon View or Report View. The items may be aligned to the top or left sides of the widget. Auto Arrange allows the widget to arrange the items itself. Edit Labels specifies whether the items are editable. The Border button specifies whether the widget should have a border. Use Images specifies whether items should display images (or just labels). In report view mode, the default sort column and direction (ascending or not) may be specified as well as the column headers.



The column header editor allows you to set up each individual header item. The **Add** button is used to add new items. The **Remove** button is used to remove items. The **Up** and **Down** buttons are used to re-order the items. For each item, you can specify its label, width and alignment. Width is expressed in pixels and may be specified as dynamic. Alignment can be left, right, center or dynamic. The image can be selected from the Bitmap Manager or specified as dynamic. Dynamic items are determined via event handlers.



## MultipleSelectListBox

*ParcPlace-Digitalk*



MultipleSelectionListBoxes are identical to single selection ListBoxes with the added ability to select multiple choices from the list.

### Protocol

#### **bitmaps**

Answer the list of bitmaps for owner drawing.

#### **bitmaps:** *aCollection*

Set the list of bitmaps for owner drawing to *aCollection*.

#### **clearSelection**

Make no list items selected

#### **contents**

Answer the list.

#### **contents:** *aCollection*

Set the contents of the widget to *aCollection*.

#### **deleteAll**

Delete the entire list.

#### **deleteIndex:** *index*

Delete item number *index* from the list.

#### **deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

**deselect**

Deselect the current selection.

**deselectAll**

Deselect all selected items in the list. Trigger the changed event.

**deselectIndex:** *itemIndex*

Deselect the item at *itemIndex* in the list. Trigger the changed event.

**deselectItem:** *anItem*

Deselect *anItem* in the list. Trigger the changed event.

**disableNoScroll**

Add the disable no scroll style.

**drawBitmap:** *aBitmap* **for:** *aString*

Use *aBitmap* in place of *aString* when displaying the contents of the list.

**extendedSelect**

Set and answer the Extended Selection List box.

**getTopIndex**

Answer the index of the first visible item.

**indexOf:** *aString*

Answer the index of *aString* in the list.

**insertItem:** *aString*

Insert the item *aString* into the list.

**insertItem:** *item* **at:** *index*

Insert the given *item* into the list (and the host control) at the given position.

**integralHeight**

Remove the no integral height style.

**itemHeight**

Answer the height of a list item.

**itemHeight:** *anInteger*

Set the height of a list item; only has an effect before the receiver window is opened.

**itemIndexFromPoint:** *aPoint*

Answer the index of the element under *aPoint* (where *aPoint* is relative to the receiver window).

**lineAt:** *index*

Answer the line at index

**list**

Answer the list in the receiver.

**list:** *anArray*

Set the list in the receiver.

**multiColumn**

Add the multi-column style.

**multipleSelect**

Add the multiple select style.

**noIntegralHeight**

Add the no integral height style.

**noRedraw**

Add the no redraw style.

**notify**

Add the notify style.

**ownerDrawFixed**

Set the style of the widget to the fixed height owner draw style.

**ownerDrawVariable**

Set the style of the widget to the variable height owner draw style.

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control; default = #printString.

**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if none is set, non-String objects are sent #printString.

**restore**

Refresh the list from the owner and maintain the position in the list without selecting it.

**restoreSelected**

Refresh the list from the owner and keep the old selection.

**restoreWithRefresh:** *anObject*

Refresh the list from the owner and keep the line containing *anObject* visible and selected.

**selectedItem**

Answer the item currently selected in the widget.

**selectedItems**

Answer a collection of the selected items.

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the index of the item currently selected in the widget.

**selection:** *anObject*

Select the line indicated by *anObject*. *anObject* may either be an index into the list or a string contained in the list.

**selections**

Answer indices of the items selected.

**setColumnWidth:** *anInteger*

Set the column width for columns in a listbox with the multiColumn style to *anInteger* pixels

**setHorizontalExtent:** *pixelWidth*

Sets the width in pixels by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will scroll items in the list box. If the list box is as large larger than this value, the horizontal scroll bar is disabled.

**setList:** *aCollection*

Set the receiver's list to aCollection. Answer *aCollection*.

**setTabStops:** *anArray*

Set the tab stop positions in the receiver to *anArray*; the receiver must have been created with the useTabStops style; tab stop positions are specified in dialog units

**setTopIndex:** *anInteger*

Set the first visible item in the receiver to be the item at *anInteger*.

**setValue:** *anItem*

Set the selection in the receiver's list to *anItem*. Trigger changed event if the selection has changed. Answer *anItem*.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger changed event if the selection has changed. Answer *itemIndex*.

**setValueIndices:** *aCollectionOfIntegers*

Set the selection to the items in *aCollectionOfIntegers*. Trigger the changed event. Answer *aCollectionOfIntegers*.

**sort**

Add the sort style.

**standard**

Add the standard style.

**useTabStops**

Add the use tab stops style.

**value**

Answer the selected item in the list.

**value:** *aCollectionOfItems*

Set the selection to the items in *aCollectionOfItems*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

**valueIndices**

Answer a collection containing the indices of the currently selected items.

**valueIndices:** *aCollectionOfIntegers*

Set the selection to the items at the index positions in *aCollectionOfIntegers*.

## Supported Events

**aboutToChange**

This event occurs whenever the selection is about to change.

**changed:** *selectedItem*

This event occurs whenever the selection is changed (passes the item as the argument).

**changedIndex:** *selectedIndex*

This event occurs whenever the selection is changed (passes the index as the argument).

**characterTyped:** *aCharacter*

This event occurs whenever any character is typed.

**clicked:** *selectedItem*

This event occurs whenever the left mouse button is clicked.

**doubleClicked:** *selectedItem*

This event occurs whenever an item has been double clicked.

**drawFocus:** *drawIndex*

This event occurs whenever the widget focus changes. The widget is rendered to indicate that it has input focus.

**drawItem:** *drawIndex*

This event occurs whenever the application receives a request to draw the widget.

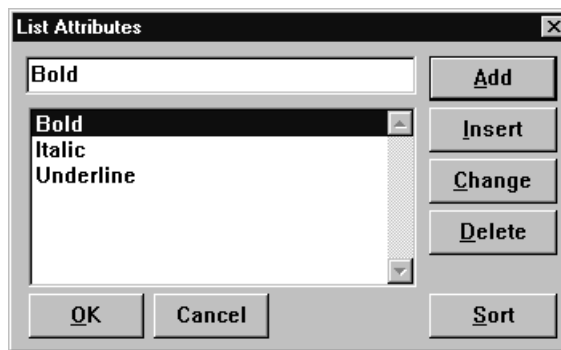
**drawSelection:** *drawIndex*

This event occurs whenever the widget receives input focus. The item is rendered to indicate that it has been selected.

**measureItem:** *drawIndex*

This event occurs whenever the height of an owner drawn item is set.

## WindowBuilder Extensions



MultipleSelectionListBoxes share the same attribute editor with ComboBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.





## ProgressBar

*ParcPlace-Digitalk - Windows 95 only*



The ProgressBar widget is a simple visual gauge.

### Protocol

#### **decrementPosition**

Decrement the current position by the current line increment.

#### **decrementPositionBy:** *anInteger*

Decrement the current position by *anInteger*.

#### **incrementPosition**

Increment the current position by the current line increment.

#### **incrementPositionBy:** *anInteger*

Increment the current position by *anInteger*.

#### **lineIncrement:** *anInteger*

Set the line increment to *anInteger*.

#### **maximum:** *anInteger*

Set the maximal value associated with the progressbar.

#### **minimum:** *anInteger*

Set the minimal value associated with the progressbar.

#### **minimum:** *minimumValue* **maximum:** *maximumValue*

Set the minimum and maximum values (range).

#### **minimum:** *minimumValue* **maximum:** *maximumValue* **position:** *positionValue*

Set the minimum and maximum values (range) and position.

#### **position**

Answer the position of the progressbar control.

**position:** *anInteger*

Set the position of the progressbar control. Answer whether the position change was successful.

**setValue:** *anInteger*

Set the position of the progressbar control to *anInteger*. Trigger the **changed** event. Answer *anInteger*.

**value**

Answer the position of the progressbar control.

**value:** *anInteger*

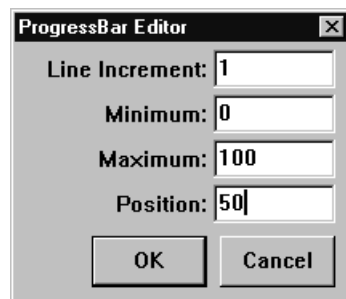
Set the position of the progressbar control to *anInteger*.

## Supported Events

**changed:** *position*

This event occurs whenever the position is changed.

## WindowBuilder Extensions



The ProgressBar attribute editor allows you to specify the line increment to associate with the widget. The minimum and maximum allowed values as well as the initial position may also be specified.



## RadioButton

*ParcPlace-Digitalk*

- ☒ **Male**
- ☐ **Female**
- ☐ **Other**

RadioButtons are used to represent mutually exclusive choices. They look and act like the radio buttons found in cars. Only one button in a group may be selected (indicated by having a dot in the center).

### Protocol

#### **autoRadioButton**

Set the automatic style. The widget will manage its own state and automatically toggle itself.

#### **click**

Programatically click the button.

#### **contents**

Set the label of the button.

#### **contents:** *aString*

Set the label of the button to the string specified by *aString*.

#### **radioButton**

Set the simple style. The developer must manage the RadioButton's state when clicked on.

#### **selection**

Answer the state of the RadioButton - true if it is on, false if off.

#### **selection:** *aBoolean*

Set the state of the RadioButton - true to check it, false to uncheck it.

#### **setLabel:** *aString*

Set the label of the receiver to *aString*. Answer *aString*.

**setValue:** *aBoolean*

Set the state of the receiver to *aBoolean*. Trigger changed event if the value has changed. Answer *aBoolean*.

**turnOff**

Set the receiver's state to off (false). Trigger changed event.

**turnOn**

Set the receiver's state to on (true). Trigger changed event.

**value**

Answer the value of the receiver as a boolean.

**value:** *aBoolean*

Set the state of the receiver to *aBoolean*.

## Supported Events

**clicked:** *labelString*

This event occurs any time the button is pressed.

**turnedOff**

This event occurs any time the widget is turned off.

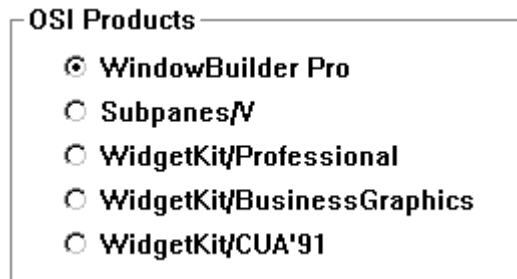
**turnedOn**

This event occurs any time the widget is turned on.



# RadioButtonGroup

*Objectshare*



RadioButtonGroup is a special CompositePane subclass that provides a mechanism for the user to select one mutually exclusive option from a set of options. RadioButtonGroups share the same protocol as ListBoxes and may be used interchangeably.

## Protocol

### **contents**

Answer the collection of labels of the RadioButtons within the group.

**contents:** *aCollectionOfStrings*

Set the labels of the RadioButtons within the group. The number of CheckBoxes is determined by the size of *aCollectionOfStrings*. **Note: This method may only be used before the window is opened.**

**indexOf:** *aString*

Answer the index of the item *aString*.

### **label**

Answer the label of the GroupBox surrounding the widget.

**label:** *aString*

Sets the text label of the GroupBox surrounding the widget.

### **list**

Return the collection of labels of the buttons within the group.

**list:** *aCollectionOfStrings*

Set the labels of the buttons within the group. The number of buttons is determined by the size of *aCollectionOfStrings*. **Note: This method may only be used before the window is opened.**

**noGroupBox**

Set the no group box style.

**numColumns**

Answer the number of columns in the group.

**numColumns:** *anInteger*

Set the number of columns in the group.

**selectedItem**

Answer the item currently selected in the widget.

**selectIndex:** *itemIndex*

Select the button at *itemIndex* in the receiver.

**selection**

Answer the index of the item currently selected in the widget.

**selection:** *anObject*

Select the line indicated by *anObject*. *anObject* may either be an index into the list or a string contained in the list.

## Supported Events

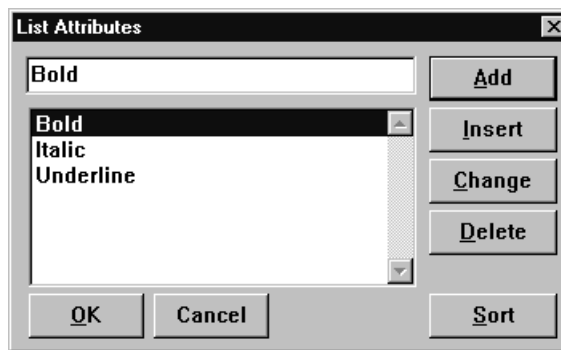
**clicked:** *aString*

This event occurs whenever one of the grouped RadioButtons is clicked.

**changedIndex:** *anIndex*

This event occurs whenever one of the grouped CheckBoxes is clicked.

## WindowBuilder Extensions

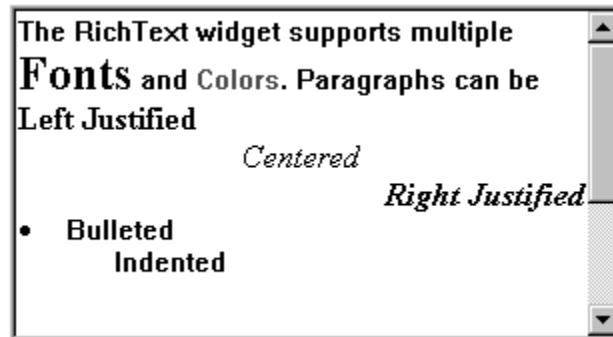


RadioButtonGroups share the same attribute editor with ListBoxes, ComboBoxes, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



# RichEdit

*ParcPlace-Digitalk - Windows 95 only*



The RichEdit widget is multi-line Rich Text (RTF) editor that supports multiple fonts, colors and formatting options.

## Protocol

### **abortChange**

Reject the new value, and restore to what was present before the user started typing.

### **append:** *aString*

Add *aString* at the end of the text in the pane.

### **autoHScroll**

Add the auto horizontal scroll style.

### **autoVScroll**

Add the auto vertical scroll style.

### **centered**

Add the centered style.

### **changeColor**

Bring up the color dialog and set the selected characters to the selected color.

### **changeFont**

Bring up the font dialog and set the selected characters to the selected font.



**changeParagraph**

Open the paragraph format dialog to change paragraph formats.

**changeTabs**

Open the change tab dialog to change tab stop positions.

**charactersBeforeLine:** *lineNumber*

Answer the number of characters preceding the given line number.

**clear**

Clear the contents of the receiver.

**clearMessage**

Answer the host message used to clear the clipboard and selection.

**clearModified**

Clear the modified flag.

**clearSelection**

Delete the current selection.

**contents**

Answer the String contained in the receiver.

**contents:** *aString*

Set the contents of the receiver to *aString*.

**copySelection**

Copy current selection to the clipboard.

**cr**

Append a line-feed to the end of the text in the pane.

**cutSelection**

Copy current selection to the clipboard then delete it from the text.

**deleteAll**

Delete all of the text in the control.

**deleteText**

Delete all of the text in the control.

**disableWordWrap**

Disable word wrap in the receiver.

**enableWordWrap**

Enable word wrap in the receiver.

**forceEndOntoDisplay**

Force the end of the text to appear on the display screen.

**forceSelectionOntoDisplay**

Force the beginning of the selection to appear on the display screen.

**formatRect**

Answer the formatting rectangle.

**formatRect:** *aRectangle*

Set the formatting rectangle to *aRectangle*.

**formattedContents**

Answer a String containing formatted contents. The inherited 'contents' method answers plain text contents

**formattedContents:** *aStreamOrString*

Set contents to formatted text from *aStreamOrString*.

**getTabStop**

Answer tabInterval in number of pels.

**getText**

Set the value instance variable from the host control's value. Assumes 'handle = NullHandle' is false.

**growTextLimit**

Increase the text limit by 32K bytes.

**insert:** *aString*

Insert *aString* at current location replacing selection if present.

**insertAfterSelection:** *aString*

Insert *aString* after the current selection and select it.

**insertAndSelect:** *aString*

Insert *aString* at current location, replacing selection if present, and select aString.

**insertSelectedText:** *aString* **at:** *anInteger*

Insert *aString* into the receiver immediately before index *anInteger* and have the newly inserted text selected.

**isModified**

Answer whether the pane has been modified. Trigger the `modifiedIsTrue` event if so.

**isReadOnly**

Answer whether the receiver is readonly.

**isTextPane**

Answer true if receiver is a kind of `TextPane`.

**isWordWrapEnabled**

Answer whether the receiver has word wrap enabled.

**leftJustified**

Add the left-justified style.

**lowerCase**

Add the lower case style.

**matchPatternBack:** *pattern* **from:** *anInteger*

Answer a `Point` representing the start and stop of the subcollection within a `Collection` that matches the receiver starting at index position *anInteger*. Answer `nil` if no match.

**maxTextHeight:** *aHeight*

Set the maximum height to be allowed for edited text.

**maxTextWidth:** *aWidth*

Set the maximum width to be allowed for edited text.

**mayUndo**

Answer true if last edit operation may be undone.

**modified**

Answer true if the receiver has been modified.

**nextPut:** *aCharacter*

Add *aCharacter* at the end of the text in the pane.

**nextPutAll:** *aString*

Add *aString* at the end of the text in the pane.

**noHideSelection**

Add the no hide selection style.

**password**

Add the password style.

**pasteSelection**

Replace selected text with the clipboard contents.

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a **changed:** handler) or the last time previousValue was sent.

**previousValue:** *aValue*

Set the previous value of the receiver.

**print**

Open a printer dialog and print the contents on the selected printer.

**printOn:** *aPrinter* **title:** *aTitle* **inset:** *insetRect*

Print the contents on *aPrinter* with *aTitle*. *insetRect* specifies the four inset margins in inches.

**readOnly**

Set the readonly property of the receiver.

**readOnly:** *aBoolean*

Set the readonly property of the receiver to *aBoolean*.

**readWrite**

Clear the readonly property of the receiver.

**restore**

Restore the contents of the receiver to the previously set contents.

**retryChange**

Stop the change process, and set the focus back to allow the user to modify the receiver and try again.

**rightJustified**

Add the right justified style.

**saveToFile:** *aPathName*

Save the contents to the file named *aPathName* in rich text format. Trigger the event

**fileException:** if any error is encountered processing the file with *aPathName*.

**selectAfter:** *position*

Move the caret to the *position* after the given character position, which may be either a Point (character in line, line) or an Integer.

**selectAll**

Select the entire text of the receiver.

**selectAtEnd**

Place the gap selection at the end of the text.

**selectBefore:** *position*

Move the caret to the position before the given character *position*, which may be either a Point (character in line, line) or an Integer.

**selectCharFrom:** *beginIndex* **to:** *endIndex*

Select from the character *beginIndex* to character *endIndex*. The indices correspond to indices of characters in the string answered by `#contents`.

**selectedItem**

Answer a String containing the text selected in clipboard format.

**selectedText**

Answer a string containing the currently selected text.

**selectFrom:** *start* **to:** *end*

Set the selection described by *start* and *end*. *start* and *end* can be either (character in line, line) pairs or character positions from beginning of the text.

**selection**

Answer a Point describing the current selection.

**selectLine:** *anInteger*

Select the entire line with the given index.

**selectLineAtChar:** *anInteger*

Select the entire line with the given character index.

**setCharColor:** *newColor*

Set the color of selected characters to *newColor*.

**setCharFont:** *aFont*

Set the font of selected characters to *aFont*.

**setContents**

Set the value of the entryfield. Assumes 'handle = NullHandle' is false.

**setFromFile:** *aPathName*

Set the contents with the text in the file named *aPathName*. Trigger the changed event.

**setModified**

Set the modified flag.

**setParagraphAlignment:** *aNumber*

Set the paragraph alignment to *aNumber* which can be PfaLeft, PfaRight, or PfaCenter.

**setParagraphOffset:** *aNumber*

Set indentation of the second and subsequent lines in the paragraph relative to the starting indentation.

**setParagraphOffsetIndent:** *aNumber*

*aNumber* is added to the start of indentation of each paragraph selected.

**setParagraphRightIndent:** *aNumber*

Set the size of the right indentation to *aNumber* which is relative to the right margin.

**setParagraphStartIndent:** *aNumber*

Set the size of the indentation in the first line of the paragraph to *aNumber*.

**setParagraphToBullet:** *aBoolean*

Set paragraph to bullet style if *aBoolean* is true.

**setTabStops:** *aList*

Set tab stop positions to *aList* in twips.

**setTextLimit:** *anInteger*

Set the maximum number of characters that the receiver can hold to *anInteger*.

**setValue:** *aValue*

Set the text contents of the receiver to the string representing *aValue*. First set the value, with a possibility that it may be undone. Then, trigger aboutToChangeTo:, where the value may be restored to the previous value, or it may be set to an entirely new value. At the end, if the final value is different from the beginning value, trigger the changed event if the value is changed. Answer the final value.

**streamContents**

Answer a ReadWriteStream containing formatted contents.

**tab**

Write a tab character to the receiver.

**tabStopInterval**

Answer tabInterval in number of pels.

**tabStopInterval:** *tabInterval*

Set tabstops to *tabInterval* number of pels.

**textLimit**

Answer the maximal number of characters allowed.

**undo**

Undo the last edit operation if possible.

**update**

Refresh the text from the owner and display it.

**upperCase**

Add the upper case style.

**value**

Answer the text contents of the receiver.

**value:** *aString*

Set the text contents of the receiver to *aString*.

**wantReturn**

Add the want return style.

**wrap**

Answer the wordwrap property of the receiver.

**wrap:** *aBoolean*

Set the wordwrap property of the receiver to aBoolean.

## Supported Events

**aboutToChangeTo:** *aString*

This event occurs when the widget's value is about to change.

**changed:** *aString*

This event occurs when the widget's value has changed.

**controlTabbed**

This event occurs when the user presses the CONTROL and TAB keys when this widget has focus.

**fileException:** *aPathName*

This event occurs when there is an error saving the contents to a file.

**modifiedIsTrue**

This event occurs when the modified flag is set to true.

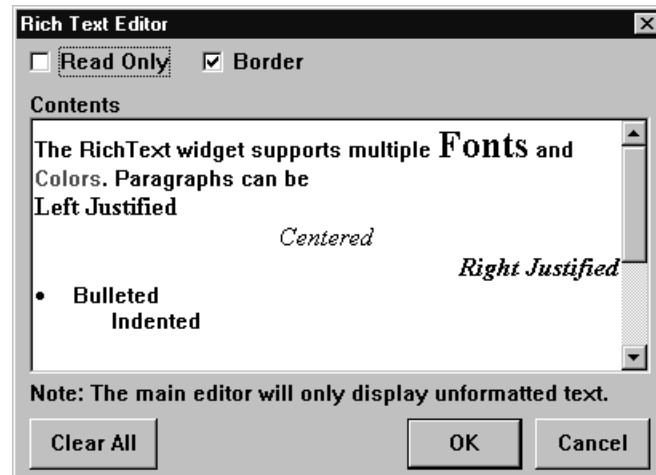
**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the #contents: method.

**textChanged:** *aString*

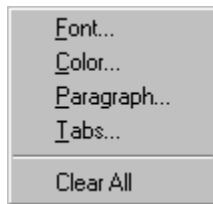
This event occurs whenever a character is typed within the field.

## WindowBuilder Extensions





The RichText attribute editor allows you to specify whether the widget is read only and whether it has a border. The initial contents of the widget may be entered in the text field. Note that the representation in the WindowBuilder Pro main editor will *not* show any formatting.



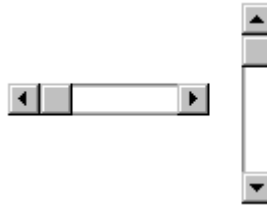
The text entry area has a popup menu attached to it from which the entered text may be modified. Multiple fonts and colors may be mixed at will. Multiple paragraph formatting options are available such as left, right or center justification, bullet points, indenting, etc. Multiple tab stops may also be specified. To clear all text and formatting information, use the Clear All command.

When initial text and formatting are supplied for the widget, the contents string for the control is generated with embedded RTF commands corresponding to any formatting options specified.



# ScrollBar

*ParcPlace-Digitalk*



ScrollBars are used as generic slider controls. They may be either vertical or horizontal. Each ScrollBar may have its own arbitrary range, line increment, and page increment.

## Protocol

**decrementPositionBy:** *anInteger*

Decrement the current position by *anInteger*.

**horizontal**

Set the horizontal style.

**incrementPositionBy:** *anInteger*

Increment the current position by *anInteger*.

**lineIncrement**

Answer the line increment.

**lineIncrement:** *anInteger*

Set the line increment to be *anInteger*.

**maximum:** *anInteger*

Set the maximum value the ScrollBar can have.

**minimum:** *anInteger*

Set the minimum value the ScrollBar can have.

**pageIncrement**

Answer the page increment.

**pageIncrement:** *anInteger*

Set the page increment to be *anInteger*.

**position**

Answer the position of the slider portion of the ScrollBar.

**position:** *anInteger*

Set the position of the slider portion of the ScrollBar.

**setScroll:** *position min: lowBound max: highBound*

Set the scrollbar position and range.

**setValue:** *anInteger*

Set the position of the slider on the scrollbar control to *anInteger*. Trigger the changed event. Answer *anInteger*.

**value**

Answer the position of the slider on the scrollbar control.

**value:** *anInteger*

Set the position of the slider on the scrollbar control to *anInteger*.

**vertical**

Set the vertical style.

## Supported Events

**changed:** *position*

This event occurs when the slider's position is changed.

**end**

This event occurs when slider is dragged to the bottom or right end.

**home**

This event occurs when slider is dragged to the top or left end.

**nextLine**

This event occurs any time the down or right arrow is clicked.

**nextPage**

This event occurs when the mouse is clicked in the area between the slider and the next arrow button.

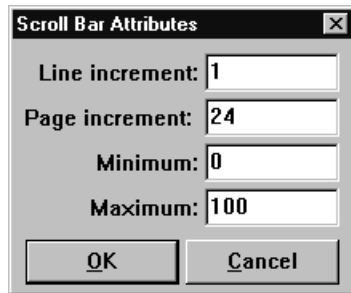
**previousLine**

This event occurs any time the up or left arrow is clicked.

**previousPage**

This event occurs when the mouse is clicked in the area between the slider and the previous arrow button.

## WindowBuilder Extensions

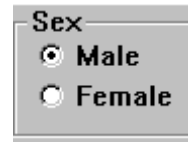


The ScrollBar attribute editor allows you to specify the line and page increments to associate with the ScrollBar. The minimum and maximum allowed values may also be specified.



## SexPane

*Objectshare*



SexPane is a special-purpose CompositePane subclass used to specify the sex of an individual. Rather than responding to individual events of its components, one need only respond to SexPane's higher-order protocol.

### Protocol

#### **contents**

Answer the sex of the widget. Allowable values are #male and #female.

#### **contents:** *aSymbol*

Set the sex of the widget. Allowable values are #male and #female.

#### **sex**

Answer the sex of the widget. Allowable values are #male and #female.

#### **sex:** *aSymbol*

Set the sex of the widget. Allowable values are #male and #female.

### Supported Events

#### **setToFemale**

This event occurs whenever the female RadioButton is clicked.

#### **setToMale**

This event occurs whenever the male RadioButton is clicked.

#### **sexChanged:** *sexSymbol*

This event occurs whenever the sex of the widget changes.

## WindowBuilder Extensions

As with all normal CompositePane classes, the attribute editor for the SexPane widget is another copy of WindowBuilder Pro. Editing the definition of SexPane class will change every instance of it.



## SpinButton

*ParcPlace-Digitalk - Windows 95 only*



SpinButtons provides a general selection capability from a collection of mutually exclusive choices. Used as an alternative to ListBoxes when space is at a premium.

### Protocol

#### **centered**

Set the window style for centered text.

#### **contents**

Answer the list contents of the receiver.

#### **contents:** *aCollection*

Set the contents of the receiver to the given list.

#### **deleteAll**

Delete the whole list.

#### **deleteIndex:** *index*

Delete item number *index* from the list.

#### **deleteItem:** *anItem*

Delete *anItem* from the receiver's list.

#### **first**

Answer the first element in the receiver.

#### **indexOf:** *aString*

Answer the index of the item *aString*.

#### **insertItem:** *aString*

Append an item containing *aString* to the receiver's list.

**insertItem:** *item at: index*

Insert the given *item* into the list (and the host control) at the given position.

**isMaster**

Answer whether the receiver is a master control.

**last**

Answer the last element in the receiver.

**leftJustified**

Set the window style for left justified text.

**lineAt:** *index*

Answer the line at *index*

**list**

Answer the list in the receiver.

**list:** *anArray*

Set the list in the receiver.

**master**

Answer the master control of the receiver.

**master:** *aSpinButton*

Set the master control of the receiver.

**maximum**

Answer the maximum value if numeric, or the size of the list if not.

**maximum:** *anInteger*

Set the maximum value of a numeric control to *anInteger*.

**minimum**

Answer the minimum value if numeric, or 1 if not.

**minimum:** *anInteger*

Set the minimum value of a numeric control to *anInteger*.

**printSelector**

Answer the selector which is sent to the items in list to format for display in the control;  
default = #printString.



**printSelector:** *aSymbol*

Set the selector which is sent to the items in list to format for display in the control; if none is set, non-String objects are sent #printString.

**restore**

Refresh the list from the owner and maintain the position in the list without selecting it.

**restoreSelected**

Refresh the list from the owner and keep the old selection.

**restoreWithRefresh:** *anObject*

Refresh the list from the owner and keep the line containing *anObject* visible and selected.

**rightJustified**

Set the window style for right justified text.

**selectedItem**

Answer the item selected in the listbox.

**selectIndex:** *itemIndex*

Select the item at *itemIndex*. Index starts at 1.

**selection**

Answer the current selection.

**selection:** *anInteger*

Set the current selection to *anInteger*.

**setList:** *aCollection*

Set the receiver's list to aCollection. Answer *aCollection*.

**setValue:** *anInteger*

Set the current selection and trigger the **changed:** event if changed.

**setValueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*. Trigger **changed:** event if the selection has changed. Answer *itemIndex*.

**spin:** *anInteger*

Spin the receiver by the amount *anInteger* (up if positive, down if negative).

**spinDown:** *anInteger*

Spin the receiver down the amount *anInteger*.

**spinUp:** *anInteger*

Spin the receiver up by the amount *anInteger*.

**value**

Answer the selected item in the list.

**value:** *anInteger*

Set the current selection to *anInteger*.

**valueIndex**

Answer the index of the selected item in the list.

**valueIndex:** *itemIndex*

Set the selection in the receiver's list to the item at *itemIndex*.

## Supported Events

**changed:** *selectedItem*

This event occurs whenever the selection is changed.

**changedIndex:** *selectedIndex*

This event occurs whenever the selection is changed.

**changing:** *selectedItem*

This event occurs whenever the selection is changing.

**changingIndex:** *selectedIndex*

This event occurs whenever the selection is changed.

**down**

This event occurs whenever the down button is clicked.

**nextLine**

This event occurs whenever the up button is clicked.

**previousLine**

This event occurs whenever the down button is clicked.

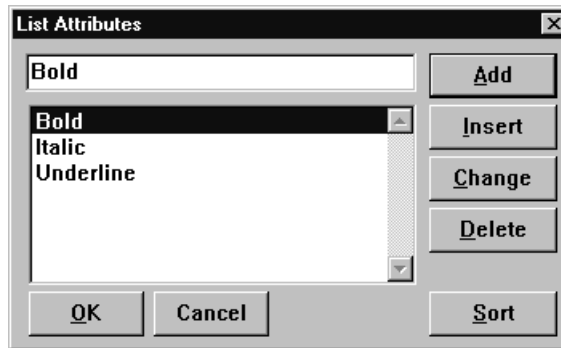
**spinnedAround:** *direction*

This event occurs whenever the spin button wraps.

**up**

This event occurs whenever the up button is clicked.

## WindowBuilder Extensions



SpinButtons share the same attribute editor with ComboBoxes, RadioButtonGroups, and other list oriented widgets. The list attribute editor provides a means to specify a list of items. Items may be added, inserted, changed, deleted, and sorted.



# StaticBox

*ParcPlace-Digitalk*



The StaticBox class is used to draw filled and outline boxes. They may either be black, gray, or white.

## Protocol

### **blackFrame**

Set the static black frame style.

### **blackRectangle**

Set the static black rectangle style.

### **grayFrame**

Set the static gray frame style.

### **grayRectangle**

Set the static gray rectangle style.

### **whiteFrame**

Set the static white frame style.

### **whiteRect**

Set the static white rectangle style.

## Supported Events

None



## StaticText

*ParcPlace-Digitalk*

**Left Justified**

**Centered**

**Right Justified**

The StaticText class provides a simple text output capability. StaticText widgets are used as labels within forms. They are often used in conjunction with EntryFields.

### Protocol

#### **centered**

Set the center justified style.

#### **contents**

Answer the contents of the widget.

#### **contents:** *aString*

Set the contents of the widget.

#### **leftJustified**

Set the left justified style.

#### **leftJustifiedNoWordWrap**

Set and answer the window style for left justified text which does not automatically wrap to the next line; this is the default style.

#### **leftJustifiedWordWrap**

Set and answer the window style for left justified text with word wrapping.

#### **noPrefix**

Add the no prefix style.

#### **rightJustified**

Set the right justified style.

**setValue: aValue**

Set the text contents of the receiver to the string representing *aValue*. Answer *aValue*.

**value**

Answer the text contents of the receiver.

**value: aString**

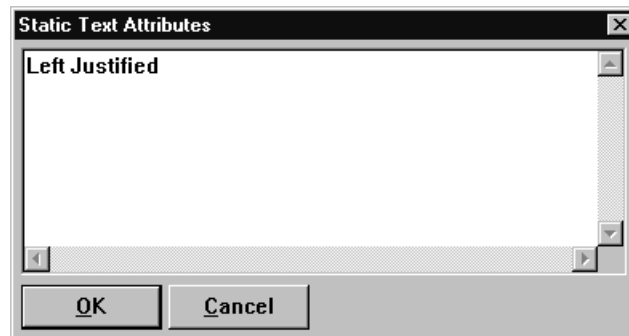
Set the text contents of the receiver to *aString*.

## Supported Events

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the #contents: method.

## WindowBuilder Extensions



The StaticText attribute editor allows you enter multi-line text for the pane.



## StatusPane

*ParcPlace-Digitalk*



The StatusPane widget provides multiple status boxes at the bottom of a window.

### Protocol

#### **contents**

Answer the receiver's contents, i.e., a collection of StatusBox objects.

**contents:** *aStatusBoxCollection*

Set the receiver's contents.

#### **fixedSize**

Set the StatusPane style. The StatusPane boxes will be fixed size.

**height:** *anInteger*

Set the receiver's height.

#### **inset**

Answer the receiver's inset. The height of a StatusBox is the height of the StatusPane minus two times the inset.

**inset:** *anInteger*

Set the receiver's inset. The height of a StatusBox is the height of the StatusPane minus two times the inset.

#### **isLeftJustified**

Answer true if the receiver is left justified, else answer false.

#### **isResizable**

Answer true if the receiver is left justified, else answer false.

#### **isRightJustified**

Set the StatusPane style - The StatusPane boxes will be rightJustified.

**justified**

Answer the justification style.

**leftJustified**

Set the StatusPane style. The StatusPane boxes will be leftJustified.

**leftJustifiedFixed**

Set the StatusPane style. The StatusPane boxes will be leftJustified and fixed width.

**resizable**

Set the StatusPane style. The left-most status box in a statusPane rightJustified will be resizable. The right-most status box in a leftJustified statusPane will be resizable.

**rightJustified**

Set the StatusPane style. The StatusPane boxes will be rightJustified.

**rightJustifiedFixed**

Set the StatusPane style. The StatusPane boxes will be rightJustified and fixed width.

**statusBoxAt:** *aSymbol*

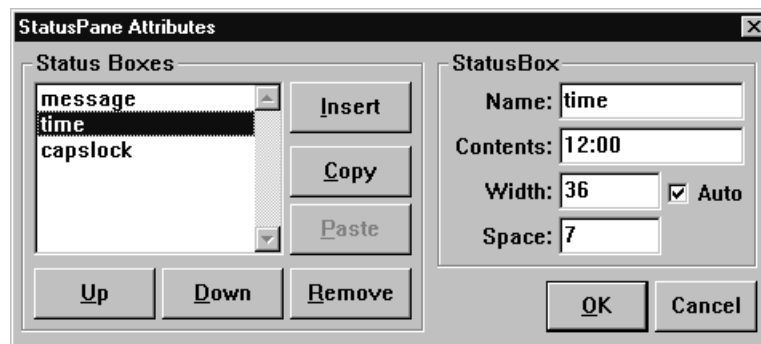
Answer the first StatusBox with the name *aSymbol*.

## Supported Events

**doubleClicked**

This event occurs whenever the widget is double clicked.

## WindowBuilder Extensions





The attribute editor for StatusPane widgets allows you to set up each individual status box. The **Insert** button is used to add new boxes. The **Remove** button is used to remove boxes. The **Up** and **Down** buttons are used to re-order the boxes. The **Copy** and **Paste** buttons are used to copy and past box definitions. For each field, you can specify its name, contents, width and spacing. Width is expressed in pixels. If the **Auto** button is checked, the width will be calculated for you as you enter the contents. Note that the **Auto** button does not apply to the widget during runtime.



# StatusWindow

*ParcPlace-Digitalk - Windows 95 only*



The StatusWindow widget provides multiple status boxes at the top or bottom of a window. Multiple box styles are supported.

## Protocol

**addField:** *aStatusField*

Add *aStatusField* to the end.

**contents**

Answer the receiver's contents, i.e., a collection of StatusField objects.

**contents:** *statusFields*

Replace the contents of the receiver with *statusFields*.

**fieldNameed:** *aSymbol*

Answer the StatusField with the name *aSymbol*.

**getBorderSizes**

Answer an Array of border horizontal size, vertical size, and gap between parts.

**height**

Answer the minimum height of the status bar.

**includeSizeGrip:** *aBoolean*

Include a size grip at the end of the status window.

**isAtTop**

Answer true if the receiver is positioned at the top of its parent window.

**placeAtTop**

Place status window at the top of the containing window.

**showHelp:** *text*

Set and display the help *text* in simple mode.

**simpleMode:** *aBoolean*

Switch the receiver to simple mode if *aBoolean* is true. The simple mode shows only one status field whose text is maintained separately from the non-simple mode fields.

**simpleModeText:** *aString*

Set simple mode text to *aString*.

**statusBoxAt:** *aSymbol*

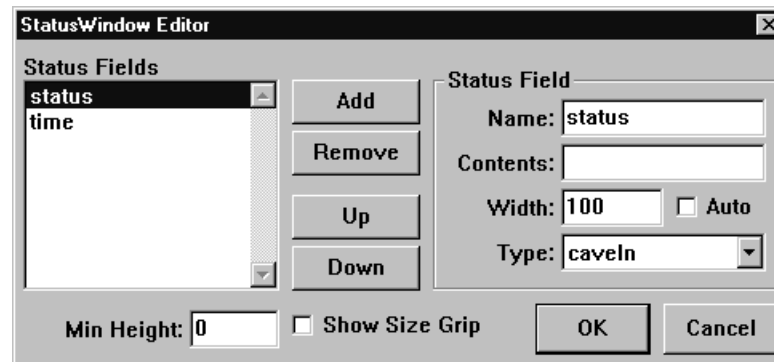
Answer the first status field with the name *aSymbol*. This method is here to be compatible with `StatusPane`.

## Supported Events

**drawItem:** *aStatusField*

This event occurs whenever an owner drawn status field needs to be drawn.

## WindowBuilder Extensions



The attribute editor for `StatusWindow` widgets allows you to set up each individual status field. The **Add** button is used to add new fields. The **Remove** button is used to remove fields. The **Up** and **Down** buttons are used to re-order the fields. For each field, you can specify its name, contents, width and type (`caveIn`, `popOut` or `noBorder`). Width is expressed in pixels. A negative width signals the field to extend itself to the right hand side of the widget. In general, you should give the last field a width of -1. If the **Auto** button is checked, the width will be calculated for you as you enter the contents. Note that the **Auto** button does not apply to the widget during runtime. The minimum height of the control may be specified as well as whether it displays a size grip in its lower right corner.



## TabControl

*ParcPlace-Digitaltalk - Windows 95 only*

The screenshot shows a window titled 'TabControl' with three tabs: 'Address', 'Name', and 'Company'. The 'Address' tab is active and contains a form with the following fields:

- Address:** A label followed by two stacked text input fields.
- City:** A text input field.
- Zip:** A text input field.
- State:** A dropdown menu.

The TabControl widget provides a multi-page notebook control with tabs. Each tab may have either a text label, an image or both. The tabs may be displayed in a single row or in multiple rows. Each page specifies a single widget for its contents (generally a CompositePane).

### Protocol

**addPage:** *aTabControlItem*

Add an item or page at the end.

**appendPage:** *aTabControlItem*

Add an item at the end.

**buttons:** *aBoolean*

Request to display tabs as buttons if *aBoolean* is true.

**contents**

Answer the collection of pages of the receiver.

**contents:** *arrayOfPages*

Set the collection of pages to *arrayOfPages*.

**currentPage**

Answer the currently selected page.

**deleteAllPages**

Delete all pages from the receiver.

**deletePage:** *aPage*

Delete *aPage* from the receiver.

**deletePageNumber:** *index*

Delete the page with the number *index*.

**demandLoad**

Answer true if demand load is requested else false.

**demandLoad:** *aBoolean*

Set the demand load flag. When this flag is set (on), the notebook page contents is not recreated until it is page is selected.

**entireClientArea**

Answer the entire client area available for children relative to origin 0@0

**firstPage**

Set the current page to the first page.

**fixedWidth:** *aBoolean*

Make all tabs the same width if *aBoolean* is true.

**freeClientArea**

Answer the remaining visible area after children affecting the available free client area have their rectangles removed from the receiver's rectangle.

**getItemHeight**

Answer tab height.

**getItemRect**

Answer tab's bounding rectangle for the first tab. This is useful when tabs are fixed size.

**getItemRect:** *tabIndex*

Answer tab size for *tabIndex*.

**getItemWidth**

Answer tab width for the first tab.

**insertPage:** *anItem at: index*

Insert *anItem* at *index*.

**insertPageAtSelection:** *anItem*

Insert *anItem* after current selected page.

**isFirstPage**

Answer true if the current item is the first one.

**isLastPage**

Answer true if the current item is the last one.

**lastPage**

Bring the last page of the receiver to the top.

**multipleLines:** *aBoolean*

Allow tabs to wrap to multiple lines if *aBoolean* is true.

**nextPage**

Bring the next item of the receiver to the top.

**ownerDraw**

Request tabs to be owner draw.

**pageCount**

Answer the number of items in the receiver.

**pageForNumber:** *index*

Answer the item object corresponding to item number *index*.

**pageNumber**

Answer the page number for the current page.

**pageNumberFor:** *anItem*

Answer the item number for *anItem*.

**pages**

Answer the collection of pages.

**previousPage**

Bring the previous page on top.

**queryFirstPage**

Answer the first item.

**queryLastPage**

Answer the last item.

**raggedRight:** *aBoolean*

Request not to align the right ends of tab lines if *aBoolean* is true.

**rectangle**

Answer the area excluding tabs.

**selectedPage**

Answer the current page.

**selectPage:** *newItem*

Programatically select a page. Will verify that the page can be selected and will trigger the events.

**setItemHeight:** *height*

Set the height of tabs to *height*.

**setItemSize:** *aPoint*

Set tab size to aPoint. Answer the old size.

**setItemWidth:** *width*

Set the width of tabs to *width*.

**setPages:** *aCollectionOfTabControlPages*

Set the pages to *aCollectionOfTabControlPages*.

**toolTips:** *aBoolean*

Request tool tips to be shown if *aBoolean* is true.

**turnToPageNumber:** *anInteger*

Turn to page number *anInteger*. If *anInteger* is out of range, do nothing.

**updateTab:** *anItem*

Update the tab of *anItem* on the screen. This is usually used after tab's attributes changed.

**wizard**

Make TabControl work like a wizard (no tabs on pages).

## Supported Events

### **aboutToChange**

This event occurs whenever a new page is about to be selected.

### **changed:** *aPage*

This event occurs whenever new page is selected.

### **changedPageNumber:** *pageNumber*

This event occurs whenever new page is selected.

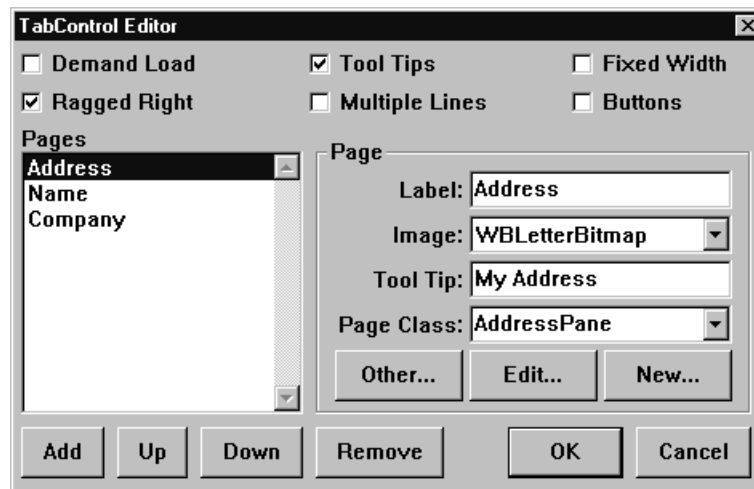
### **drawItem:** *index*

This event occurs whenever an tab needs to be drawn.

### **inputOccurred**

This event occurs whenever input occurs on any of the child widgets.

## WindowBuilder Extensions



The attribute editor for TabControl widgets allows you to set up each page. The **Add** button is used to add new pages. The **Remove** button is used to remove pages. The **Up** and **Down** buttons are used to re-order the pages. For each page, you can specify its label, image, tool tip and page class (generally a CompositePane). Widgets other than CompositePanes may be selected via the **Other** button. The **Edit** button opens another copy of WindowBuilder on the selected CompositePane page class. the **New** button launches another copy of WindowBuilder in order to create a new CompositePane class.





# TextEdit

*ParcPlace-Digitalk*



TextEdits are used to provide multi-line text entry and edit capabilities. If the user types more text than can be accommodated within the field, it will automatically scroll.

## Protocol

### **autoHScroll**

Add the auto horizontal scroll style.

### **autoVScroll**

Add the auto vertical scroll style.

### **centered**

Add the centered style.

### **clear**

Clear the contents of the receiver.

### **clearSelection**

Delete the current selection.

### **contents**

Answer the contents of the field as a string.

### **contents:** *aString*

Set the contents of the field to the string *aString*.

### **copySelection**

Copy current selection to the clipboard.

**cutSelection**

Copy current selection to the clipboard then delete it from the text.

**deleteAll**

Delete all of the text in the control.

**deleteText**

Delete all of the text in the control.

**disableWordWrap**

Disable word wrap in the receiver.

**enableWordWrap**

Enable word wrap in the receiver.

**formatRect**

Answer the formatting rectangle.

**formatRect:** *aRectangle*

Set the formatting rectangle to *aRectangle*.

**getTabStop**

Answer tabInterval in number of pels.

**insert:** *aString*

Insert *aString* at current location replacing selection if present.

**insertSelectedText:** *aString* **at:** *anInteger*

Insert *aString* into the receiver immediately before index *anInteger* and have the newly inserted text selected.

**leftJustified**

Add the left-justified style.

**lowerCase**

Add the lower case style.

**maxTextHeight:** *aHeight*

Set the maximum height to be allowed for edited text.

**maxTextWidth:** *aWidth*

Set the maximum width to be allowed for edited text.

**mayUndo**

Answer true if last edit operation may be undone.

**modified:** *aBoolean*

Set whether the field has been modified.

**nextPut:** *aCharacter*

Add a character at the end of the text in the pane.

**nextPutAll:** *aString*

Add *aString* at the end of the text in the pane.

**noHideSelection**

Add the no hide selection style.

**oemConvert**

Add the OEM convert style.

**password**

Add the password style.

**pasteSelection**

Replace selected text with the clipboard contents.

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a #changed: handler) or the last time previousValue was sent.

**previousValue:** *aValue*

Set the previous value of the receiver.

**readOnly**

Set the readonly property of the receiver.

**readOnly:** *aBoolean*

Set the readonly property of the receiver to *aBoolean*.

**readWrite**

Clear the readonly property of the receiver.

**retryChange**

Stop the change process, and set the focus back to allow the user to modify the receiver and try again.

**rightJustified**

Add the right justified style.

**selectAfter:** *position*

Move the caret to the position after the given character *position*, which may be either a Point (character in line, line) or an Integer.

**selectAll**

Select all of the text in the field.

**selectAtEnd**

Place the gap selection at the end of the text.

**selectBefore:** *position*

Move the caret to the position before the given character *position*, which may be either a Point (character in line, line) or an Integer.

**selectedItem**

Answer a String containing the text selected in clipboard format.

**selectFrom:** *aStartIndex* **to:** *anEndIndex*

Select the specified range of text.

**setTextLimit:** *anInteger*

Set the maximum number of characters that the receiver can hold to *anInteger*.

**setValue:** *aValue*

Set the text contents of the receiver to the string representing *aValue*. First set the value, with a possibility that it may be undone. Then, trigger `aboutToChangeTo:`, where the value may be restored to the previous value, or it may be set to an entirely new value. At the end, if the final value is different from the beginning value, trigger the changed event if the value is changed. Answer the final value.

**tabStopInterval**

Answer `tabInterval` in number of pels.

**tabStopInterval:** *tabInterval*

Set tabstops to `tabInterval` number of pels.

**undo**

Undo the last edit operation if possible.

**upperCase**

Add the upper case style.

**value**

Answer the text contents of the receiver.

**value:** *aString*

Set the text contents of the receiver to *aString*.

**wantReturn**

Add the want return style.

**wrap**

Answer the value of the word-wrap attribute.

**wrap:** *aBoolean*

Set the wordwrap property of the receiver to *aBoolean*.

## Supported Events

**aboutToChangeTo:** *aString*

This event occurs when the widget's value is about to change.

**changed:** *aString*

This event occurs when after the widget's value has changed.

**entered:** *aString*

This event occurs when the user presses the Enter key when this widget has focus.

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the `#contents:` method.

**textChanged:** *aString*

This event occurs whenever a character is typed within the field.



# TextPane

*ParcPlace-Digital*



TextPanes are used to provide multi-line text entry and edit capabilities. If the user types more text than can be accommodated within the field, it will automatically scroll. In some VisualSmalltalk. platforms, TextPanes are implemented entirely in Smalltalk as opposed to TextBoxes which are real operating system widgets.

## Protocol

### **contents**

Answer the contents of the field as a string.

### **contents:** *aString*

Set the contents of the field to the string *aString*.

### **cr**

Append a line-feed to the text in the field.

### **expressionEvaluator**

Answer the instance of ExpressionEvaluator (or subclass) which handles evaluations.

### **expressionEvaluator:** *anExpressionEvaluator*

Set the instance of ExpressionEvaluator (or subclass) which handles evaluations.

### **fileInFrom:** *aFileStream*

Refresh the pane data with the current contents of *aFileStream*.

### **fileOutOn:** *aFileStream*

Write the pane data out on *aFileStream*.

### **forceEndOntoDisplay**

Force the end of the text to appear on the display screen.

**forceSelectionOntoDisplay**

Force the origin of the selection to appear on the display screen.

**insert:** *aString*

Insert *aString* at current location replacing selection if present.

**insertAfterSelection:** *aString*

Insert *aString* after the current selection and select it.

**insertAndSelect:** *aString*

Insert *aString* at current location, replacing selection if present, and select *aString*.

**modified:** *aBoolean*

Change modified to *aBoolean*.

**nextPut:** *aCharacter*

Add *aCharacter* to the end of the text.

**nextPutAll:** *aString*

Add *aString* to the end of the text.

**previousValue**

Answer the previous value of the receiver. The previous value is the value prior to any user changes (when there is a #changed: handler) or the last time previousValue was sent.

**previousValue:** *aValue*

Set the previous value of the receiver.

**readOnly**

Make the control a read only control.

**readWrite**

Make the control a read and write control.

**restore**

Restore the contents of the receiver to the previously set contents.

**selectAfter:** *aPoint*

Place the selection after *aPoint*.

**selectAll**

Select all of the text in the field.

**selectAtEnd**

Place the selection at the end of the text.

**selectBefore:** *pointOrInteger*

Place the selection before *pointOrInteger*.

**selectedItem**

Answer the currently selected text.

**selectFrom:** *start* **to:** *end*

Select the rectangle with origin *start* and extent *end*.

**setEvaluate:** *aBoolean*

Specify whether a compilation for the receiver is evaluating an expression or compiling a method.

**setValue:** *aValue*

Set the text contents of the receiver to the string representing *aValue*. Trigger the changed event if the value is changed. Answer *aValue*.

**topCorner:** *aPoint*

Change the top-left corner of the receiver pane to *aPoint* which causes the text located below *aPoint* to be displayed.

**value**

Answer the text contents of the receiver.

**value:** *aString*

Set the text contents of the receiver to *aString*.

## Supported Events

**aboutToSave**

This event occurs when the widget's contents are about to be saved.

**changed:** *aString*

This event occurs when after the widget's value has changed.

**needsContents**

This event occurs when the widget first comes up. This is a good opportunity to set the widget's contents using the `#contents:` method.



**saved**

This event occurs after the widgets contents have been saved.



# ThreeStateButton

*ParcPlace-Digitalk*

☒ **Bold**

☐ **Italic**

☒ **Underline**

ThreeStateButtons are almost exactly like CheckBoxes except that they may be in an indeterminate state (neither checked or unchecked). The widget will Answer either **true** or **false** if it is checked or unchecked and will Answer **nil** if in an indeterminate state. These buttons not only allow the user to set or change their values, but they also act as an indicator of the value's current state.

ThreeStateButtons are often used in the case that a set of selected objects do not share the same value of some characteristic.

## Protocol

### **autoCheckBox**

Set and answer the automatic check box window style.

### **autoThreeState**

Set the automatic style. The widget will manage its own state and automatically toggle itself.

### **check**

Set the receiver's state to checked (true). Trigger changed event.

### **checkBox**

Set and answer the check box window style.

### **click**

Programatically click the button.

### **contents**

Answer the label of the button.

### **contents:** *aString*

Set the label of the button to the string specified by *aString*.

**indeterminate**

Set the receiver's state to indeterminate (filled). Trigger changed event.

**label:** *aString*

Set the label of the receiver to *aString*.

**selection**

Answer the state of the ThreeStateButton - true if it is on, false if off and nil if it is indeterminate.

**selection:** *aBooleanOrNil*

Set the state of the widget - true to check it, false to uncheck it, nil to become indeterminate.

**threeState**

Set the simple style. The developer must manage the widget's state when clicked on.

**setLabel:** *aString*

Set the label of the receiver to *aString*. Answer *aString*.

**setValue:** *aBoolean*

Set the state of the receiver to *aBoolean*. Trigger changed event if the value has changed. Answer *aBoolean*.

**threeState**

Set and answer the style for three state buttons.

**uncheck**

Set the receiver's state to unchecked (false). Trigger changed event.

**value**

Answer the value of the receiver as a boolean.

**value:** *aBoolean*

Set the state of the receiver to *aBoolean*.

## Supported Events

**clicked:** *aBooleanOrNil*

This event occurs any time the button is pressed.

**checked**

This event occurs any time the widget is turned on.

**indeterminate**

This event occurs any time the widget is set to its indeterminate state.

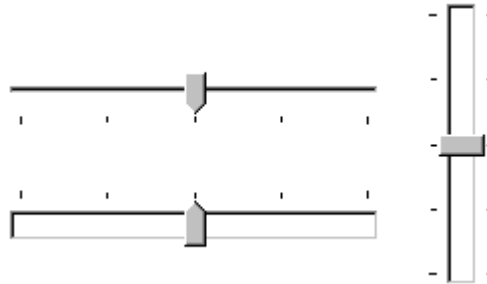
**unchecked**

This event occurs any time the widget is turned off.



## TrackBar

*ParcPlace-Digitalk - Windows 95 only*



The TrackBar widget is a slider control supporting multiple scales and tick marks. It can have scales and tick marks on either or both sides.

### Protocol

**addTick:** *anInteger*

Add a tick mark at the given position (which is a position within the current range).

**decrementPosition**

Decrement the current position by the current line increment.

**decrementPositionBy:** *anInteger*

Decrement the current position by *anInteger*.

**enableRangeSelection**

Add the range selection enabled style.

**horizontal**

Set the horizontal style.

**incrementPosition**

Increment the current position by the current line increment.

**incrementPositionBy:** *anInteger*

Increment the current position by *anInteger*.

**isHorizontal**

Answer whether the receiver is horizontal style.

**isVertical**

Answer whether the receiver is vertical style.

**lineIncrement**

Answer the line increment.

**lineIncrement:** *anInteger*

Set the line increment to be *anInteger*.

**maximum**

Answer the maximal value associated with the trackbar.

**maximum:** *anInteger*

Set the maximal value associated with the trackbar.

**minimum**

Answer the minimal value associated with the trackbar.

**minimum:** *anInteger*

Set the minimal value associated with the trackbar.

**minimum:** *minimumValue* **maximum:** *maximumValue*

Set the minimum and maximum values (range).

**minimum:** *minimumValue* **maximum:** *maximumValue* **position:** *positionValue*

Set the minimum and maximum values (range) and position.

**pageIncrement**

Answer the page increment.

**pageIncrement:** *anInteger*

Set the page increment to be *anInteger*.

**position**

Answer the position of the slider on the trackbar control.

**position:** *anInteger*

Set the position of the slider on the trackbar control. Answer whether the position change was successful.

**setValue:** *anInteger*

Set the position of the slider on the scrollbar control to *anInteger*. Trigger the **changed** event. Answer *anInteger*.

**ticks**

Answer the Array of tick mark positions.

**ticks:** *anArray*

Set the Array of tick mark positions to *anArray*.

**ticksBoth**

Set the style which indicates that tick marks are displayed on both left and right (vertical) or top and bottom (horizontal).

**ticksBottom**

Set the style which indicates that tick marks are displayed on the bottom (horizontal).

**ticksLeft**

Set the style which indicates that tick marks are displayed on the left (vertical).

**ticksRight**

Set the style which indicates that tick marks are displayed on the right (vertical).

**ticksTop**

Set the style which indicates that tick marks are displayed on the top (horizontal).

**value**

Answer the position of the slider on the scrollbar control.

**value:** *anInteger*

Set the position of the slider on the scrollbar control to *anInteger*.

**vertical**

Set the vertical style.

## Supported Events

**changed:** *position*

This event occurs whenever the position is changed.

**changing:** *position*

This event occurs whenever the position is changing.

**end**

This event occurs when slider is dragged to the bottom or right end.

**home**

This event occurs when slider is dragged to the top or left end.

**nextLine**

This event occurs any time the down or right arrow is clicked.

**nextPage**

This event occurs when the mouse is clicked in the area between the slider and the next arrow button.

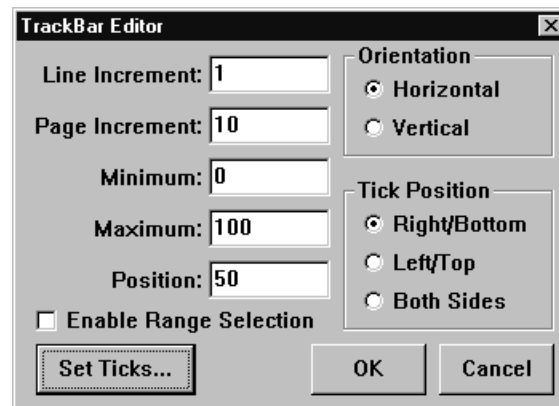
**previousLine**

This event occurs any time the up or left arrow is clicked.

**previousPage**

This event occurs when the mouse is clicked in the area between the slider and the previous arrow button.

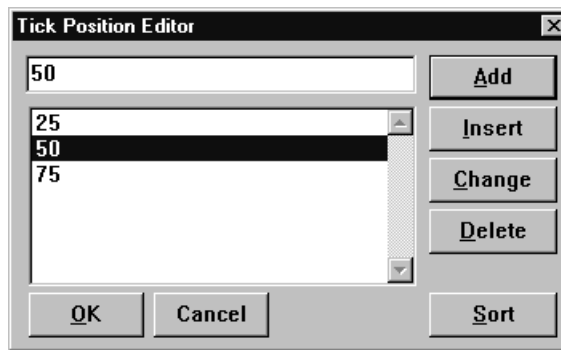
## WindowBuilder Extensions



The TrackBar attribute editor allows you to specify the line and page increments to associate with the widget. The minimum and maximum allowed values as well as the initial position may also be specified. The orientation may be either horizontal or vertical. The **Enable Range Selection** button causes the control to display a thick shaft.

The **Set Ticks** button launches the Tick Position Editor. Ticks may appear on either side or both sides.



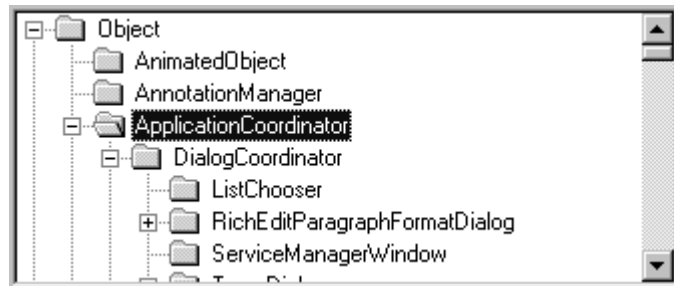


Ticks may be added, inserted, changed, deleted, and sorted.



# TreeView

*ParcPlace-Digitalk - Windows 95 only*



The TreeView widget is a hierarchical list control with expand and collapse capabilities. Optional plus/minus buttons and images may be displayed.

## Protocol

**collapseItem:** *anObject*

Collapse the children of *anObject*.

**contents**

Answer the contents of the receiver.

**contents:** *aCollection*

Set the contents of the receiver to *aCollection*.

**deleteAll**

Delete the whole list.

**deleteItem:** *anObject*

Delete *anObject* from the items in the receiver.

**editItemLabel:** *anObject*

Begin direct edit for the specified item.

**editLabels:** *aBoolean*

Enable or disable label editing, according to the value of *aBoolean*.

**ensureVisible:** *anObject*

Make anObject (an item) visible within the receiver.

**expandItem:** *anObject*

Expand the children of *anObject*.

**getChildrenExpandedFor:** *aTreeViewItem*

Get the children expanded for *aTreeViewItem*.

**getChildrenFor:** *aTreeViewItem*

Get the children for *aTreeViewItem*.

**getHasChildrenFor:** *aTreeViewItem*

Answer whether the *aTreeViewItem* has children.

**getImageFor:** *aTreeViewItem*

Get the image for *aTreeViewItem*.

**getSelectedImageFor:** *aTreeViewItem*

Get the selected image for *aTreeViewItem*.

**getStringFor:** *aTreeViewItem*

Get the label for *aTreeViewItem*.

**hasButtons:** *aBoolean*

Determine whether the receiver has buttons for expanding and collapsing items, according to the value of *aBoolean*.

**hasLines:** *aBoolean*

Determine whether the receiver has lines, according to the value of *aBoolean*.

**hasLinesAtRoot:** *aBoolean*

Set whether the receiver has lines at the root level, according to the value of *aBoolean*.

**indent**

Answer the indent width (pixels) of the receiver.

**indent:** *anInteger*

Set the indent width (pixels) of the receiver to anInteger.

**itemsAttribute**

Answer the contents of the receiver.

**selectedItem**

Answer the currently selected item.

**selection**

Answer the currently selection.

**selection:** *anObject*

Set the current selection to *anObject*.

**selectItem:** *anObject*

Set the current selection to *anObject*.

**showSelectionAlways:** *aBoolean*

Determine whether the receiver displays its selection even when not active, according to the value of *aBoolean*.

**useImages**

Answer whether any specified images in the list contents are to be used in displaying the list items (default is true if images have been supplied).

**useImages:** *aBoolean*

Determinate whether any specified images in the list contents are to be used in displaying the list items (default is true if images have been supplied).

**value**

Answer the current selection.

## Supported Events

**aboutToChangeLabel:** *anItem to: newLabel*

This event occurs whenever an item's label is about to change.

**aboutToEditLabel:** *anItem*

This event occurs whenever an item's label is about to be edited.

**changed:** *anItem*

This event occurs whenever the selection is changed.

**changedLabel:** *anItem to: newLabel*

This event occurs whenever an item's label is changed.

**characterTyped:** *aCharacter*

This event occurs whenever a character is typed.

**collapsed:** *anItem*

This event occurs whenever an item is collapsed.

**doubleClicked:** *anItem*

This event occurs whenever an item is double clicked.

**expanded:** *anItem*

This event occurs whenever an item is expanded.

**needsChildrenFor:** *anItem*

This event occurs whenever an item's children are required.

**needsHasChildrenFor:** *anItem*

This event occurs whenever an item's parent state is required.

**needsImageFor:** *anItem*

This event occurs whenever an item's image is required.

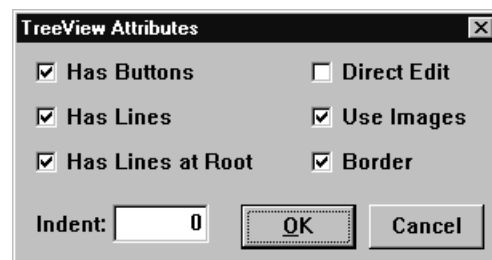
**needsSelectedImageFor:** *anItem*

This event occurs whenever an item's selected image is required.

**needsStringFor:** *anItem*

This event occurs whenever an item's label is required.

## WindowBuilder Extensions



The TreeView attribute editor allows you to specify whether the widget displays plus and minus buttons and whether there are lines displayed. The control may also allow direct editing. Whether each item displays an image may be specified as well as whether the widget has a border. Finally, the indent between levels may be specified.



# UpDown

*ParcPlace-Digitalk - Windows 95 only*



The UpDown widget is a simple control that is a composite between an up and a down button.

## Protocol

### **decrementPosition**

Decrement the current position by the current line increment.

### **decrementPositionBy:** *anInteger*

Decrement the current position by *anInteger*.

### **horizontal**

Set the horizontal style.

### **incrementPosition**

Increment the current position by the current line increment.

### **incrementPositionBy:** *anInteger*

Increment the current position by *anInteger*.

### **isHorizontal**

Answer whether the receiver is horizontal style.

### **isVertical**

Answer whether the receiver is vertical style.

### **lineIncrement**

Answer the line increment.

### **maximum**

Answer the maximal value associated with the updown.

**maximum:** *anInteger*

Set the maximal value associated with the updown.

**minimum**

Answer the minimal value associated with the updown.

**minimum:** *anInteger*

Set the minimal value associated with the updown.

**minimum:** *minimumValue* **maximum:** *maximumValue*

Set the minimum and maximum values (range).

**minimum:** *minimumValue* **maximum:** *maximumValue* **position:** *positionValue*

Set the minimum and maximum values (range) and position.

**position**

Answer the position of the updown control.

**position:** *anInteger*

Set the position of updown control. Answer whether the position change was successful.

**setValue:** *anInteger*

Set the position of the updown control to *anInteger*. Trigger the **changed** event. Answer *anInteger*.

**value**

Answer the position of the updown control.

**value:** *anInteger*

Set the position of the updown control to *anInteger*.

**vertical**

Set the vertical style.

**wrap**

Answer whether wrapping is enabled.

**wrap:** *aBoolean*

Turn wrapping on or off, according to *aBoolean*.

## Supported Events

**changed:** *position*

This event occurs whenever the position is changed.

**changing:** *position*

This event occurs whenever the position is changing.

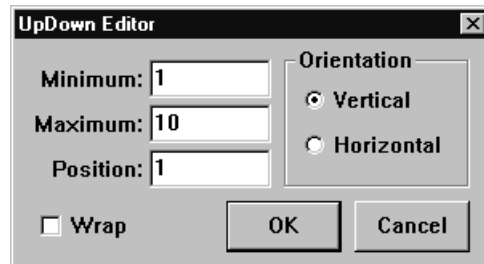
**nextLine**

This event occurs any time the down or right arrow is clicked.

**previousLine**

This event occurs any time the up or left arrow is clicked.

## WindowBuilder Extensions



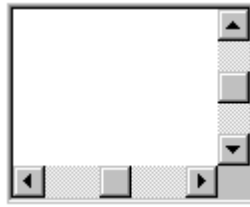
The UpDown attribute editor allows you to specify the minimum and maximum allowed values as well as the initial position. The orientation may be either horizontal or vertical. The **Wrap** button causes values to wrap from minimum to maximum.





## VideoPane

*ParcPlace-Digitalk - Windows 95 only*



VideoPane provides a simple multimedia control.

### Protocol

#### **contents**

Answer the contents of the receiver.

#### **contents:** *aVideo*

Set the contents of the receiver to *aVideo*.

#### **stretch**

Answer whether the receiver stretches its contents to fit the receiver's current size.

#### **stretch:** *aBoolean*

Set whether the receiver stretches its contents to fit the receiver's current size, according to the value of *aBoolean*.

#### **video**

Answer the contents of the receiver.

#### **video:** *aVideo*

Set the contents of the receiver to *aVideo*.

### Supported Events

**None**



# WBStaticGraphic

*Objectshare*



WBStaticGraphic is a replacement for Digitalk's StaticGraphic widget. It allows you to display a bitmap image within a window.

## Protocol

### **contents**

Answer the bitmap displayed by the widget.

### **contents:** *aBitmap*

Set the contents of the widget to the bitmap specified by *aBitmap*.

### **fixedSize**

If the widget has a bitmap as its label, draw the bitmap at its normal size.

### **stretchToFit**

If the widget has a bitmap as its label, stretch or shrink the bitmap to fill the widget.

## Supported Events

**None**

## WindowBuilder Extensions

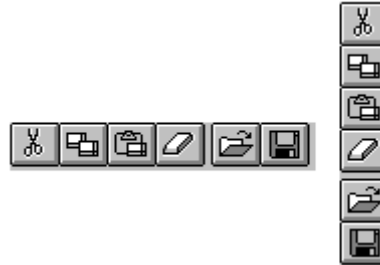


The attribute editor for WBStaticGraphics allows you to select a bitmap file (.BMP) from disk.



# WBToolBar

Objectshare



WBToolBar provides a general purpose horizontal or vertical tool bar control similar to that found in dozens of commercial applications. Each button may optionally specify selectors for the left and right mouse buttons as well as the spacing between it and its neighbor.

## Protocol

**add:** *aBitmapOrSelector*

Add a button to the toolbar. It will not have a right button selector.

**add:** *aBitmapOrSelector* **rbSelector:** *rbSelector*

Add a button to the toolbar. Its right button selector is *rbSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **enable:** *enableSelector*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **enable:** *enableSelector* **owner:** *owner*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*. The owner is *owner*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **enable:** *enableSelector* **owner:** *owner*  
**spaces:** *numSpaces*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*. The owner is *owner*. It will be *numSpaces* from its neighbor.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **enable:** *enableSelector* **owner:** *owner*  
**spaces:** *numSpaces* **rbSelector:** *rbSelector*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*. The owner is *owner*. It will be *numSpaces* from its neighbor. Its right button selector is *rbSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **owner:** *owner*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **owner:** *owner* **spaces:** *numSpaces*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The enable selector is *enableSelector*. It will be *numSpaces* from its neighbor.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **owner:** *owner* **spaces:** *numSpaces*  
**rbSelector:** *rbSelector*

Add a button to the toolbar. Its left button selector is *aSelector*. It will not have a right button selector. The owner is *owner*. It will be *numSpaces* from its neighbor. Its right button selector is *rbSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **rbSelector:** *rbSelector*

Add a button to the toolbar. Its right button selector is *rbSelector*.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **spaces:** *numSpaces*

Add a button to the toolbar. It will be *numSpaces* from its neighbor. It will have no right button selector.

**add:** *aBitmapOrSelector* **selector:** *aSelector* **spaces:** *numSpaces* **rbSelector:** *rbSelector*

Add a button to the toolbar. Its left button selector is *aSelector*. It will be *numSpaces* from its neighbor and its right button selector is *rbSelector*.

**add:** *aBitmapOrSelector* **spaces:** *numSpaces*

Add a button to the toolbar. It will be *numSpaces* from its neighbor. It will have no right button selector.

**add:** *aBitmapOrSelector* **spaces:** *numSpaces* **rbSelector:** *rbSelector*

Add a button to the toolbar. It will be *numSpaces* from its neighbor and its right button selector is *rbSelector*.

**cellHeight**

Answer the height of the buttons.

**cellSize**

Answer the size of the buttons.

**cellSize:** *aPoint*

Set the size of the buttons. The default is 25@22.

**cellWidth**

Answer the width of the buttons.

**currentIndex**

Answer the number of the button currently touched by the mouse. The buttons are number from left to right starting at one.

**disableElements**

Disable all of the buttons.

**disableItem:** *aSelector*

Disable the button with the left button selector *aSelector*.

**enableElements**

Enable all of the buttons.

**enableItem:** *aSelector*

Enable the button with the left button selector *aSelector*.

**postAutomatic**

Set the postAutomatic style. This will cause the buttons to remain pressed until the action they initiate finishes.

**preAutomatic**

Set the postAutomatic style. This will cause the buttons to Answer to their unpressed states before the action they initiate takes place.

**rbSelectorAt:** *aKey*

Answers the right button selector of the button with the index *aKey*.

**selectedItem**

Answers the index of the selected button.

**selectItem:** *aKey*

Select button with the index *aKey*.

**selector**

Answers the left button selector of the selected button.

**selectorAt:** *aKey*

Answers the left button selector of the button with the index *aKey*.

**toggle**

Set the toggle style. This will cause the buttons to remain pressed until another one is pressed.

**vertical**

Set the vertical style.

## Supported Events

**clicked:** *selector*

This event occurs whenever a button is selected.

**doubleClicked:** *selector*

This event occurs whenever a button is double clicked.

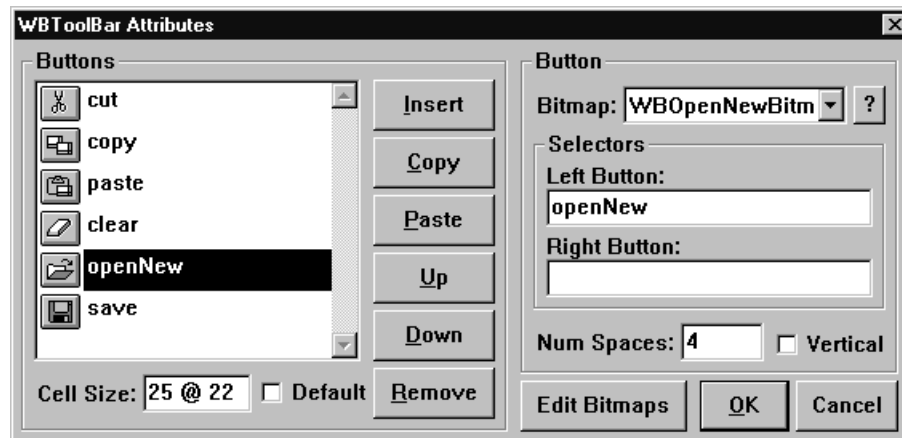
**selecting:** *selector*

This event occurs whenever an enabled button is pressed while the mouse is down within the toolbar. A select event may also be generated if the mouse is released while in the button.

**showHelp:** *selector*

This event occurs whenever an button is touched while the mouse is down within the toolbar. This event occurs regardless of whether the button is enabled.

## WindowBuilder Extensions



The WBToolBar attribute editor provides everything you need to create and edit toolbars. The ListBox on the left side of the window shows the buttons that make up the toolbar and their corresponding left button selectors.

Below the list, the **Cell Size** field provides a place where the size of the buttons may be specified. For normal applications the default is 25@22. The **Default** CheckBox will set the cell size to the default and disable the cell size field.

The buttons to the right of the list are used to manipulate the toolbar buttons. The **Insert** button will insert a new button after the currently selected button. The **Copy** button will copy the current button. The **Paste** button will paste the previously copied button after the current button. The **Up** button will move the current button up while the **Down** button will move it down. Finally, the **Remove** button will delete the currently selected button.

On the right side of the editor can be found controls for manipulating the currently selected toolbar button. The **Bitmap** combobox can be used to select a toolbar button from the Bitmap Manager. *Only bitmaps designed to be used as toolbar buttons should be used.* Toolbar button bitmaps are defined with their up and down states side by side. They should be exactly twice as wide as the cell size used by the toolbar. If this is not the case, the toolbar buttons will not look very good. Fortunately, WindowBuilder Pro provides a mechanism for automatically generating button templates and editing them (See the section on the Bitmap Manager for further details).

Below the bitmap selection combobox are fields for specifying the **Left Button** and **Right Button** selectors to be used with the button. The left button selector will always initially default to the same name as the button. Ideally you would give your toolbar



bitmaps names that correspond to the functions they are meant to invoke. The right button selector is optional and is generally used to popup a menu for the button.

Below these selectors is the **Num Spaces** field used to specify how far (in pixels) the currently selected button should be from the one before it. The default is zero which will result in buttons that are right next to one another.

The **Vertical** checkbox is used to rotate the toolbar so that it displays vertically (horizontal is the default).

The **Edit Bitmaps** button at the bottom of the editor invokes the Bitmap Manager. From there you may create and edit any bitmaps. It provides special commands for specifically creating and editing toolbar bitmaps.





---

# Chapter 11 Windows and Dialogs

WindowBuilder Pro provides specific support for windows and dialogs in addition to widgets. In this section of the manual, we discuss several window-oriented protocols and events that you can use with WindowBuilder Pro. We will also describe the window and dialog attribute editors.

## Protocol

### **abortClose**

Veto the window close which is about to occur.

### **activeTextPane**

Answer the subpane that has the typing focus.

### **addDialogBorderStyle**

Add the dialog border style.

### **addMaximizeButtonStyle**

Add the maximize button style.

### **addMaximizeStyle**

Add the maximize style.

### **addMinimizeButtonStyle**

Add the minimize button style.

### **addMinimizeStyle**

Add the minimize style.

### **addModalBorderStyle**

Add the modal border style.

### **addSizingBorderStyle**

Add the sizing border style.

**addSubpane:** *aSubPane*

Add *aSubpane* to the receiver.

**addSubpaneDynamically:** *aSubPane*

Add *aSubpane* to a window dynamically.

**addSystemMenuStyle**

Add the system menu style.

**addSystemModalStyle**

(dialogs only)

Add the system modal style.

**addTitleBarStyle**

Add the title bar style.

**backColor**

Answer the background color of the receiver.

**backColor:** *aColor*

Set the background color of the receiver.

**bringToFront**

Make the receiver the front window.

**changed:** *aFacet*

Something has changed related to the dependents of the receiver. Send update message to all the dependents of *aFacet* (usually the name of a subpane).

**changed:** *aFacet* **with:** *firstParameter*

Something has changed related to the dependents of the receiver. Send update: *firstParameter* message to all the dependents of *aFacet* (usually the name of a subpane).

**changed:** *aFacet* **with:** *firstParameter* **with:** *secondParameter*

Something has changed related to the dependents of the receiver. Send the 'update: *firstParameter* with: *secondParameter*' to all the dependents of *aFacet* (usually the name of a subpane).

**childAtId:** *anInteger*

Answer the child with the id *anInteger*.

**children**

Answer the collection of the receiver's children.

**childrenSize**

Answer the size of all descendents.

**cursorWindowPosition**

Answer the current position of the cursor in window coordinates.

**disable**

Disable the receiver and all its children. A disabled window does not receive user input.

**disabled**

Answer true if the receiver is disabled, else false.

**disableRedraw**

Disable redrawing of the receiver.

**disableUpdate**

Disable update of the receiver and all its children.

**enable**

Enable the receiver and all its children.

**enableRedraw**

Allow the receiver to be redrawn; force the receiver to repaint itself.

**enableUpdate**

Enable update of the receiver and all its children.

**foreColor**

Answer the foreground color of the receiver.

**foreColor:** *aColor*

Set the foreground color of the receiver.

**frameRectangle**

Answer the receiver's frame area as a rectangle.

**frameRelativeRectangle**

Answer the receiver's frame window rectangle relative to the super window.

**hasSmalltalkMenuBar**

Answers whether the receiver should have the default Smalltalk menu bar.

**height**

Answer the receiver's height.

**hideWindow**

Make the window and all its children invisible.

**icon**

Answer the receiver's icon.

**icon:** *anIcon*

Set the receiver's icon

**initialSize:** *aSize*

Set the initial window size to *aSize*.

**label**

Answer the receiver's label.

**label:** *aString*

Set the window label of the receiver to *aString*.

**labelWithoutPrefix:** *aString*

Set the window label without the WindowLabelPrefix of the receiver to *aString*.

**maximumSize**

Answer the maximum size that the receiver may be resized to with the sizing border.

**maximumSize:** *aPoint*

Set the maximum size that the receiver may be resized to with the sizing border.

**menuTitled:** *aString*

Answer the menu from the receiver's menu bar whose title is *aString*. Case is ignored and *aString* can include '&', '~', or neither.

**minimumSize**

Answer the minimum size that the receiver may be resized to with the sizing border.

**minimumSize:** *aPoint*

Set the minimum size that the receiver may be resized to with the sizing border.

**noSmalltalkMenuBar**

Does not create the default Smalltalk menu bar (File, Edit, Smalltalk menus). By sending this message, the owner takes the responsibility for creating the menu bar upon reception of #menuBuilt event. Menu specific to SubPanes, (built with #getMenu event) will not be added to the MenuBar but will only be added to the popup menu.

**openWindow**

Open the receiver.

**rectangle**

Answer the bounding rectangle of the receiver.

**rectangle:** *aRectangle*

Set the position of the receiver to *aRectangle*.

**redraw**

Redraw the entire window.

**removeMaximizeButtonStyle**

Remove the maximize button style.

**removeMinimizeButtonStyle**

Remove the minimize button style.

**removeSizingBorderStyle**

Remove the sizing border style.

**removeSubpane:** *aWindow*

Remove *aWindow* from the receiver.

**removeSubpaneDynamically:** *aSubPane*

Remove *aSubpane* from a window dynamically.

**removeSystemMenuStyle**

Remove the system menu style.

**startTimer:** *idInteger* **period:** *millisecondsInteger*

Start a timer identified by *idInteger*. A #wmTimer:with: message will be sent to the receiver every *millisecondsInteger*.

**stopTimer:** *idInteger*

Stop the timer identified by *idInteger* for aWindow.



**subPaneWithFocus**

Answer the SubPane of the receiver that last had the focus

**timerID**

Answer the id of the timer that is being notified, during the #timer event. Else answer nil.

**toggleWrap**

Word wrap was selected from the Options menu; pass on the request to the active TextPane (TextPaneControl).

**width**

Answer the receiver's width.

**windowPolicy**

Answer the receiver's instance of WindowPolicy or a subclass which is used to construct the menu bar.

**windowPolicy:** *aWindowPolicy*

Set the receiver's instance of WindowPolicy or a subclass which is used to construct the menu bar.

## Supported Events

**aboutToClose**

This event occurs whenever a window is about to close. Closing can be vetoed by #abortClose.

**activated**

This event occurs whenever a window becomes active. The active window has the input focus and displays its border using the active window border color. Note that a window becomes active when the user clicks on it *and* when it is first opened by the system.

**closed**

This event occurs whenever a window has been closed.

**deactivated**

This event occurs after a window has been deactivated. A window is deactivated before another window is activated.

**drawFocus:**

This event occurs whenever focus needs to be drawn for an owner-drawn menu item.

**drawItem**

This event occurs whenever an owner-drawn menu item is drawn.

**drawSelection:**

This event occurs whenever an owner-drawn menu item is selected.

**help**

This event occurs whenever the F1 key is pressed. You can handle this event yourself and display your own help windows or you can let the operating system handle it.

**maximized**

This event occurs whenever a window is maximized.

**measureItem**

This event occurs whenever an owner-drawn menu item's height must be provided.

**menuBarBuilt**

(windows only)

This event occurs after the menu bar has been built.

**minimized**

This event occurs whenever a window is minimized.

**opened**

This event occurs after the window has been opened and validated.

**resized**

This event occurs whenever the window is resized.

**restored**

This event occurs whenever the window is restored (from a minimized state).

**rightClicked**

This event occurs whenever the right button is clicked in the window

**timer: *timerID***

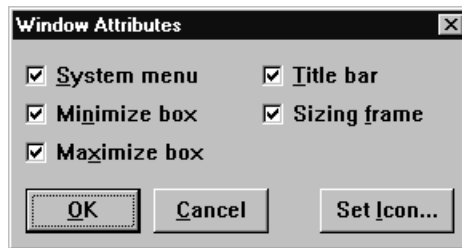
This event occurs whenever the window receives a time message from the operating system. For more information on setting up timers, refer to Digitalk's documentation.

**validated**

(windows only)

This event occurs after the window has become a true operating system window but before its subpanes have been built or it has been displayed to the user.

## WindowBuilder Extensions

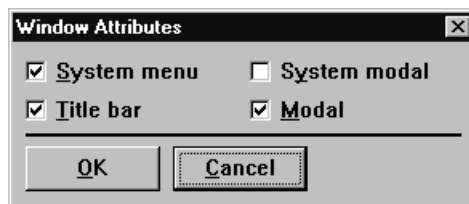


The Window attribute editor allows you to specify the frame characteristics of the window. A window may optionally have a system menu, a minimize box, a maximize box, a title bar, or a sizing frame.

Clicking the **Set Icon...** button will open a file dialog from which you may choose an icon file (.ICO) to attach to the window. This is the icon that will be used when the window is minimized.



WindowBuilder Pro provides an easy way to attach icons to windows. There are other mechanisms to do this if you want to access icons from resource files. See Digitalk's documentation for further details.



The Dialog attribute editor allows you to specify the frame characteristics of a dialog. A dialog may optionally have a system menu or a title bar. The dialog box may also be modal (to the active window) or a system modal (modal to all of the application's windows).

# Chapter 12 CompositePanels

This section discusses how to create and write code for CompositePanels. CompositePanels can be virtually any combination of widgets or other composite panels.

## Creating Composite Panels

New compositepanels may be created in one of two ways. The File menu, New CompositePanel command will create an empty compositepanel to which you may add widgets. Alternatively, you may select a collection of widgets in the edit window and then use the File menu, CompositePanel | Create command (this command is also available from the popup group menu).

Editing the contents of a compositepanel is handled like editing a window or dialog. Widgets may be added and event handlers may be specified for them.

When you are done specifying the compositepanel it may be saved with the normal File menu, Save command. All compositepanels are subclasses of CompositePanel or one of its subclasses. This is directly analogous to window and dialogs. Rather than a `#createView` method, compositepanels define an `#addSubpanels` method. The `#addSubpanels` method defines the layout and contents of the compositepanel.

Event handling method stubs will also be created. You should flesh them out in the same way you would complete a window or dialog event handling stub.

## Styles

CompositePanels may be displayed in several styles. The default style causes them to display without a border or any scrollbars. The `borders` style results in a box around the compositepanel. The `scrollbars` style gives the compositepanel vertical and horizontal scrollbars. The `verticalScrollBar` style adds a single vertical scrollbar.

## Nesting

CompositePanels may be nested arbitrarily deep. Very powerful interface objects can be created by building up layers of compositepanels.

**Warning:** be careful not to define a compositepane to include a copy of itself. WindowBuilder Pro will stop you from adding a compositepane to itself, but it cannot check for deeper levels of recursion (for example, A contains B which contains A).

## Tab Order

The tab order that should be in effect within the compositepane may also be specified. When compositepane are nested or when they are used in a window this tab order is properly maintained (compositepanes acts like single objects in the Tab editor of their containing windows).

## Adding Events

When creating a compositepane it is often useful to define events that are specific to the compositepane that can be exported to its owner (generally the window in which it resides). This also provides an opportunity to rename the real events to something more meaningful.

Adding events to a compositepane is simple. First define a class method for the compositepane called `#constructEventsTriggered`. This method should call `super constructEventsTriggered` and then add its own events (defined as Symbols). An example from the `OkCancelPane` example provided with WindowBuilder Pro would look like:

```
constructEventsTriggered
  ^super constructEventsTriggered
    add: #ok;
    add: #cancel;
    yourself.
```

This example exports two events, `ok` and `cancel`. Actually generating these events is the second step. Generating an event is easy. The code is of the form `self triggerEvent: #eventname`. Looking at the `OkCancelPane` example again, we see that it converts `clicked` events on its two buttons to `ok` and `cancel` events for its owner to use.

For example, if the OK button in the `OkCancelPane` performs the message `#ok` when it receives a `clicked` event, the code for the `ok` method would look like:

```
ok
  self triggerEvent: #ok
```

The owner window could then respond to the `ok` event of the `OkCancelPane` directly rather than a `clicked` event on the button itself.

For further details on adding your own events to CompositePanels, it is beneficial to examine some of the other CompositePanel examples provided with WindowBuilder Pro.



## Chapter 13 Pool Managers

WindowBuilder Pro provides very sophisticated pool dictionary management functions. These functions make it very easy to share resources across multiple window definitions and easy to support applications that must run at different resolutions (and which need *different* resource mappings at each resolution). Functions are provided for efficiently swapping pool dictionary values at runtime. This makes it easy to swap languages for an application using National Language Support (NLS) or to swap font definitions when switching resolutions (e.g., small fonts under 640x480 and large fonts under 1024x768).

In WindowBuilder Pro, there are three types of pool managers: Bitmap Pool Manager, Font Pool Manager, and NLS Pool Manager. Each pool manager ‘owns’ and manages a collection of pools. Each pool ‘owns’ and manages a global dictionary (the actual global pool dictionary in the Smalltalk dictionary) and a collection of sub-pools where each sub-pool represents a ‘category’. The global pool dictionary and each of the sub-pools share the same set of keys. The values in the ‘active’ sub-pool (active category) are the same as the values in the global pool dictionary.

In general, each pool dictionary has multiple pool variables (key/value pairs) in it. Using the pool managers in WindowBuilder Pro, each pool dictionary has multiple categories, with each category/key combination having one value associated with it. The pool variables contain the value associated with the active category.

Each pool manager contains three lists and an edit area. The list of managed pools appears in the upper left and the list of categories appears in the upper right. The list of keys contained in the selected pools appears in the lower left, while the lower right contains the ‘edit area’ specific to each type of pool manager. When exactly one category is selected, then the list of keys will display the key name and the value for that category/key pair. When exactly one key is selected, the list of categories will display the name of the category and the value for that category/key pair.



# General Menu Definition

## Pool

### New

Creates a new pool to be managed.

### Add Existing

Adds an existing pool dictionary to the list of managed pools. The existing pool must contain homogeneous values. For example, when adding an existing pool to the bitmap manager, the existing pool must contain only bitmaps as values before the pool can be managed.

### Update Dependent Classes

Recompiles any classes whose definition includes the selected pools.

### Remove

Remove the pool from the list of managed pools. This does *not* delete the pool dictionary from the Smalltalk system dictionary.

### Delete

Remove the pool from the list of managed pools *and delete* the pool dictionary from the Smalltalk system dictionary.

## Category

### New

Add a new category to the list of categories. The values for each of the new category/key pairs are copied from the active category/key pairs. The New Category dialog is shown in Figure 13-1.



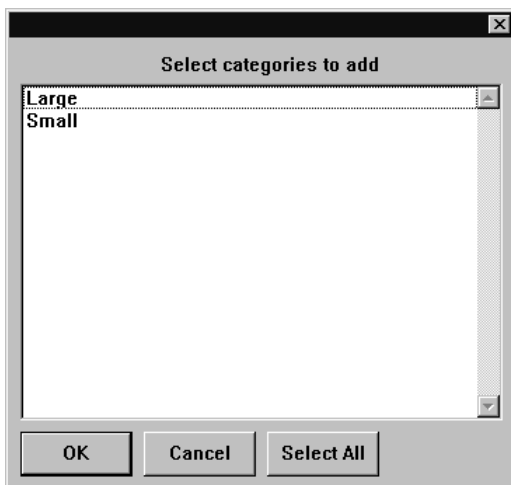
**Figure 13-1** New Category dialog.

### Rename

Rename the currently selected category.

### Add Suggested

Opens a dialog displaying a list of suggested categories that can be added. Any categories that are highlighted when the OK button is pressed are added to the list of categories in the pool manager window. The Add Suggested Category dialog is shown in Figure 13-2.



**Figure 13-2** Add Suggested Category dialog.

**Delete**

Deletes the selected categories from the list of categories. All values associated with that category are deleted as well

**Show All**

When checked, all categories for all managed pools are visible in the category list

**Show Selected**

When checked, only those categories in the currently selected pools are visible in the category list

**Active**

This menu contains a list of all categories, with a check beside the active category. When a category is selected as the active category, then for each managed pool, the value for each key in that category is copied into the global pool dictionary associated with that pool.

**Key****New**

Add a new key to the currently selected pool

**Rename**

Rename the currently selected key

**Move**

Move the currently selected keys from one pool to another

**Browse References**

Open a method browser on all methods referencing the selected key

**Delete**

Delete the currently selected key

## Option

### **Show Active Category**

When checked, the active category is displayed in a combobox above the list of categories in the pool manager window

### **Show Path**

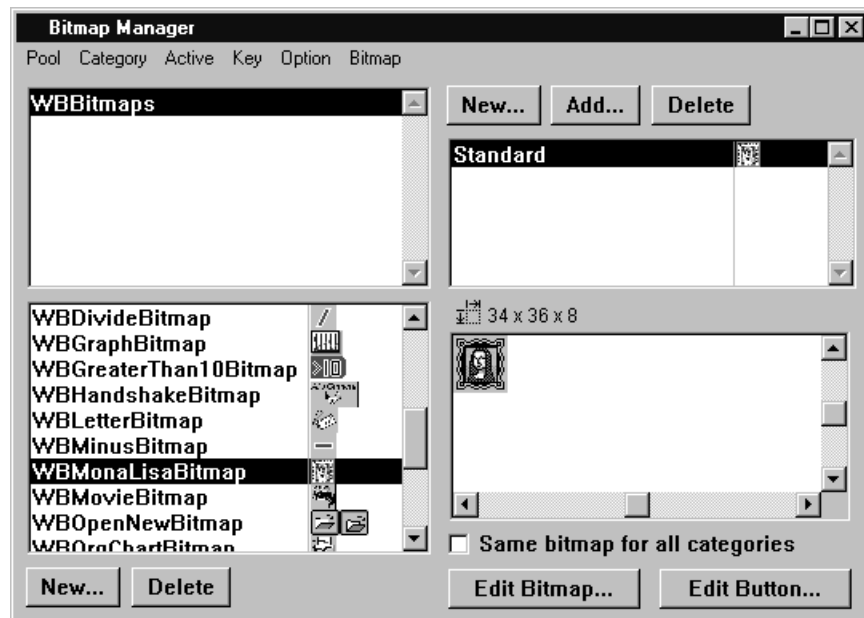
When checked, the file path for both the selected pool and the selected category are shown below the pool list and the category list respectively

### **Allow Duplicate Keys**

When checked, a key in one managed pool may have the same name as a key in another managed pool. It is recommended that unique keys be used for all managed pools.

## Bitmap Manager

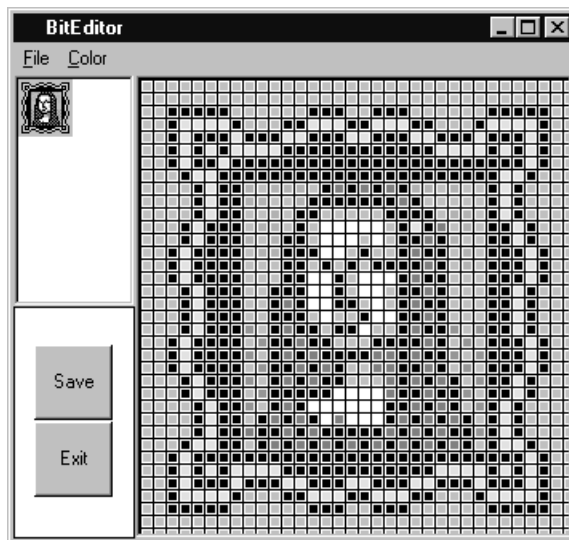
The Bitmap Manager shown in Figure 13-3 is a convenient place to create, edit, store and retrieve bitmaps from within the VisualSmalltalk environment. The upper left, upper right and lower left panes are generic and act as described in the first section. In addition to the names, the key and category lists display thumbnail views of the bitmaps. The edit pane in the lower right corner displays the full bitmap plus information on its size (width, height and color depth).



**Figure 13-3** Bitmap Manager.

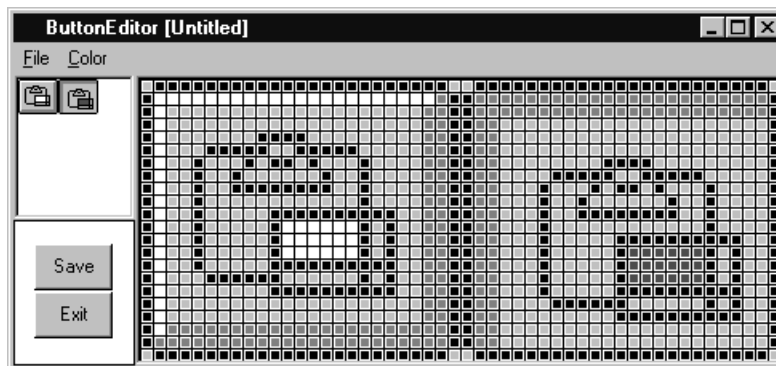
Clicking on the size button (next to the dimensions) brings up a dialog from which you can resize the bitmap. The checkbox below the image display forces the displayed bitmap to be the same across all categories.

The Edit Bitmap button launches the bitmap editor as shown in Figure 13-4. This is a very simple bitmap editor which allows you to set individual bits in the image to any of the 16 standard colors. It also supports flood-filling an area with a specific color.



**Figure 13-4** Bitmap Editor.

The Edit Button button launches the button editor as shown in Figure 13-5. This editor is specialized to the task of editing button bitmaps where the left side of the bitmap represents the 'up' version and the right side represents the 'down' side. As you place pixels in the left half of the button, these pixels are offset and mirrored on the right side. Color fills do not cause this to happen. After editing the left side, you can make subtle alterations to the right side to indicate the depressed state.



**Figure 13-5** Button Editor.

There are several file formats for bitmaps: BPL (Bitmap Pool file format), BDT (Bitmap Dictionary file format), and BMP (BitMap file format). The BMP file format is the standard window bitmap file format that the Microsoft Paint program can edit. Individual bitmaps can be read from and written to files in BMP format.

The BDT file format is the older bitmap dictionary file format and is supported for compatibility with older versions of WindowBuilder Pro. This file format contains multiple key/value pairs but does not contain any information indicating what pools and categories contain these key/value pairs.

The BPL file format contains complete information (pool names, category names, key names, and bitmaps or bitmap references) so the complete pool and categories can be rebuilt when they are filed in.

## Pool Menu

### File In

File in a BPL file, creating the pools, keys, and/or categories specified in the file if they do not already exist. If a BDT file is selected, then the keys are placed in the currently selected pool and currently selected category. To load a BPL file programatically (i.e., when constructing a runtime image), use the following expression:

```
WBBitmapManager current load: 'BITMAPS.BPL'
```

### File Out

File out a single BPL file containing all categories/keys/bitmaps for the currently selected pools. Multiple pools may be filed out to a single BPL file and will be correctly recreated when the BPL file is loaded back into the image.

## Category Menu

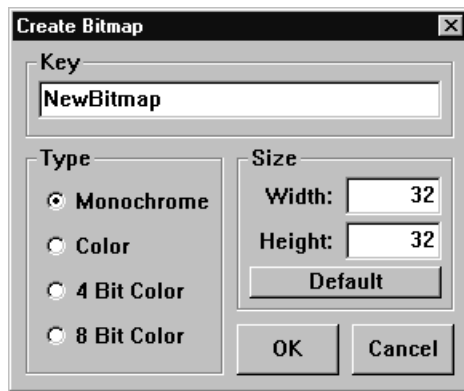
### File Out

File out a single BPL file containing all keys/bitmaps for the currently selected categories in the currently selected pools. Multiple pools and multiple categories may be filed out into a single BPL file and will be correctly recreated when the BPL file is loaded back into the image.

## Key Menu

### New

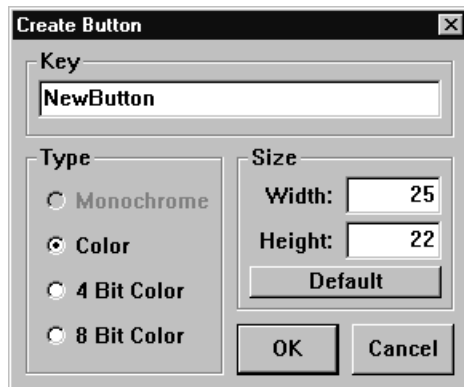
Opens the dialog shown in Figure 13-6. This dialog prompting the user for the key name, the bitmap type (color, monochrome) and the bitmap size. Clicking OK will create a new key in the currently selected pool in the currently selected category with the specified bitmap type and size. Once created, use the 'Edit Bitmap' button located in the bottom right of the pool manager window to edit the new bitmap.



**Figure 13-6 Create Bitmap Dialog.**

### New Button

Similar to New, this command creates a new key/bitmap with the bitmap in a format that can be used with a WBToolBar. Once created, use the 'Edit Button' button located in the bottom right of the pool manager window to edit the new button. The Create Button Dialog is shown in Figure 13-7.



**Figure 13-7 Create Button Dialog.**

### Duplicate

Create a new bitmap based on existing ones. It will prompt you for a new key then create a copy of the currently selected bitmap with that key.

### From Clipboard

Create a new key/bitmap based on the bitmap in the clipboard. It will prompt you for a new key then create a copy of the bitmap contained in the clipboard.



### **From Screen**

Create a new key/bitmap from an area of the screen. It will prompt you for a new key, then change the cursor to indicate that you should click and drag across a rectangular area of the screen. A new key/bitmap will be created based on the name and screen area specified.

### **File In Bitmap File**

Create a new key/bitmap from a BMP file. It will prompt you for a new key, then for the name of a BMP file. A new key/bitmap will be created based on the name and BMP file specified.

### **File Out BDT File**

File out the currently selected keys/bitmaps in BDT file format. The pool and category information will NOT be stored in the BDT file.

### **File Out RC and Supporting Files**

#### **File Out RC and 4 Bit Supporting Files**

#### **File Out RC and 8 Bit Supporting Files**

These menu options generate \*.RC and \*.BMP files that can be used by Microsoft Visual C++ to generate a Dynamic Link Library (DLL) containing the specified bitmaps. The File Out RC and 4 Bit Supporting Files and File Out RC and 8 Bit Supporting Files menu commands converts each bitmap to 4 bit and 8 bit respectively before it is saved in its BMP file.

### **Resize**

Resize the current bitmap. This command opens a dialog to specify the new bitmap dimension and whether or not the image should be cropped or scaled.

### **Resize Button**

Intelligently resize the current button. This command opens a dialog to specify the new bitmap button dimensions and whether or not the image should be cropped or scaled. The Bitmap/Button Resize Dialog is shown in Figure 13-8.

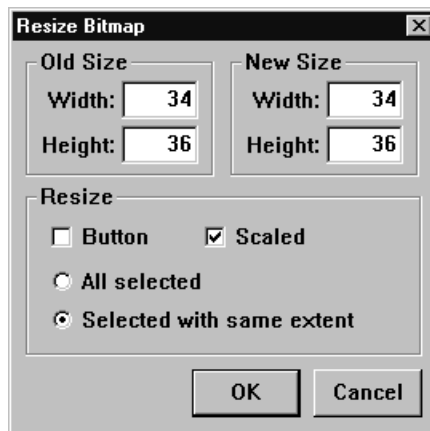


Figure 13-8 Resize Bitmap/Button Dialog.

## Option Menu

### Same Bitmap All Categories

When checked, all new key/bitmap pairs that are created will have the same exact bitmap object for each category.

### Show Dimensions

When checked, display the bitmap width, height, and bit count in the key list and the category list.

### Update Source

When checked and Team/V is present, update the pool variable definition of all bitmaps stored in DLLs whenever the bitmap for that key is changed.

## Bitmap Menu

### Copy

Copy the current bitmap to the clipboard

### Paste

Paste the bitmap in the clipboard replacing the current bitmap

**Paste Into Button**

Paste the bitmap in the clipboard into both the up and down areas of the current bitmap button

**Paste Into Button Up**

Paste the bitmap in the clipboard into the up area of the current bitmap button

**Paste Into Button Down**

Paste the bitmap in the clipboard into the down area of the current bitmap button

**From Screen**

Replace the current bitmap with a bitmap generated from a specified area of the screen. This command changes the cursor to indicate that a rectangular area of the screen should be specified using the click and drag of the mouse. The current bitmap is then replaced with a new bitmap based on the specified area of the screen.

**From Screen Into Button**

Replace the up and down areas of the current bitmap button with a bitmap generated from a specified area of the screen. This command changes the cursor to indicate that a rectangular area of the screen should be specified using the click and drag of the mouse. The current bitmap is then replaced with a new bitmap based on the specified area of the screen.

**From Screen Into Button Up**

Replace the up area of the current bitmap button with a bitmap generated from a specified area of the screen. This command changes the cursor to indicate that a rectangular area of the screen should be specified using the click and drag of the mouse. The current bitmap is then replaced with a new bitmap based on the specified area of the screen.

**From Screen Into Button Down**

Replace the down area of the current bitmap button with a bitmap generated from a specified area of the screen. This command changes the cursor to indicate that a rectangular area of the screen should be specified using the click and drag of the mouse. The current bitmap is then replaced with a new bitmap based on the specified area of the screen.

### **File In Bitmap File**

Replace the current bitmap with a bitmap from a BMP file.

### **File Out Bitmap File**

File out the current bitmap into a BMP file.

### **File Out 4 Bit Bitmap File**

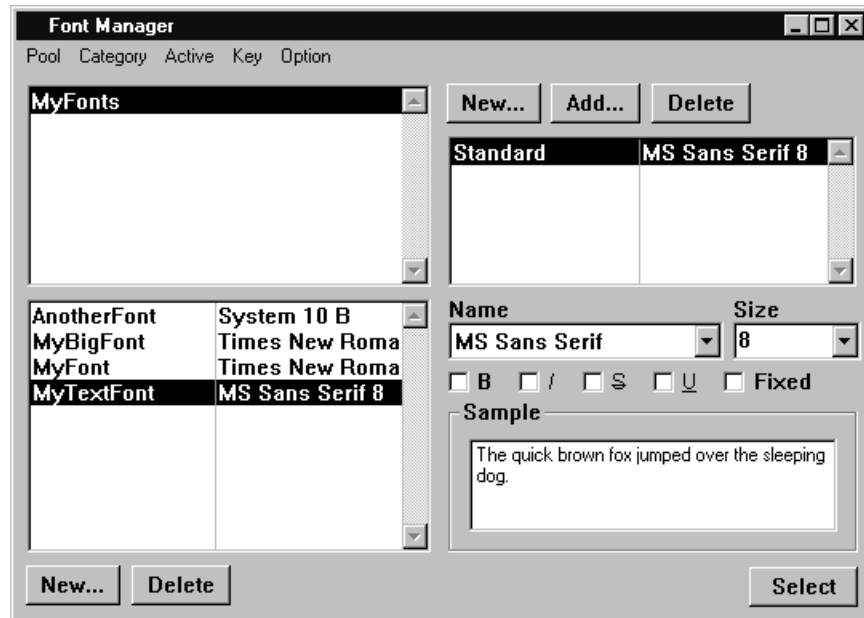
File out the current bitmap in a 4 bit BMP file.

### **File Out 8 Bit Bitmap File**

File out the current bitmap into an 8 bit BMP file.

## Font Manager

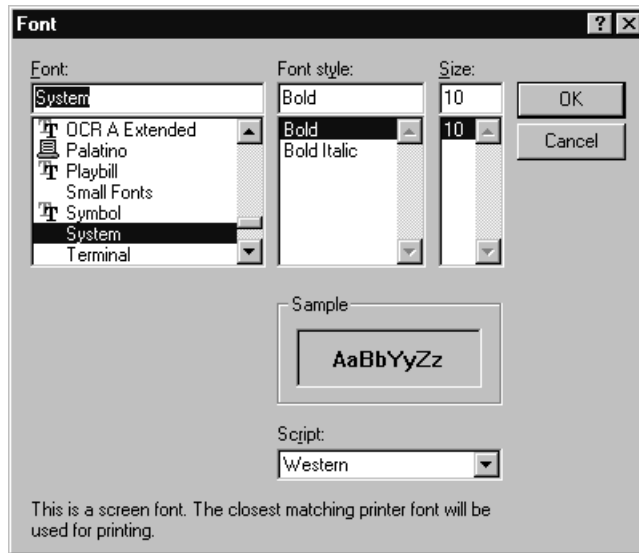
The Font Manager shown in Figure 13-9 is a convenient place to create, edit, store and retrieve named fonts from within the VisualSmalltalk environment. The upper left, upper right and lower left panes are generic and act as described in the first section. In addition to the names, the key and category lists display descriptions of the fonts. The edit pane in the lower right corner displays the full information on the font (face name, size, bold, italic, strikethrough, underline, fixed) as well as a sample of the actual font.



**Figure 13-9** Font Manager.

Fonts may be specified by selecting their various attributes from the comboboxes and checkboxes. If a specified combination does not exist in the system, the system will attempt to use the closest match. Alternatively, you can select a font from the standard font dialog by clicking on the Select button. The standard font dialog is shown in Figure 13-10.

Fonts are stored in FPL (font pool file format) formatted files. These files contain pool names, category names, keys, and font definitions. When the FPL file is loaded into the image, the pools, categories, keys, and font will be created as they were when they were saved.



**Figure 13-10 Standard Font Editor.**

Fonts may be filed out in portable or non-portable format. When filed out in portable format, the font name, size, and characteristics are stored as strings. When file out in non-portable format, the entire font is represented as a byte array.

## Pool Menu

### File In

This command prompts the user to select a FPL file to be loaded into the image. To load a FPL file programatically (i.e., when constructing a runtime image), use the following expression:

```
WBFontManager current load: 'MYFONTS.FPL'
```

### File Out

This command files out all categories, keys, and font definitions of the currently selected pool into an FPL file.

## Category Menu

### File Out

This command files out all keys and font definitions in the currently selected categories and currently selected pools into an FPL file.

## Option Menu

### **Portable File Out Format**

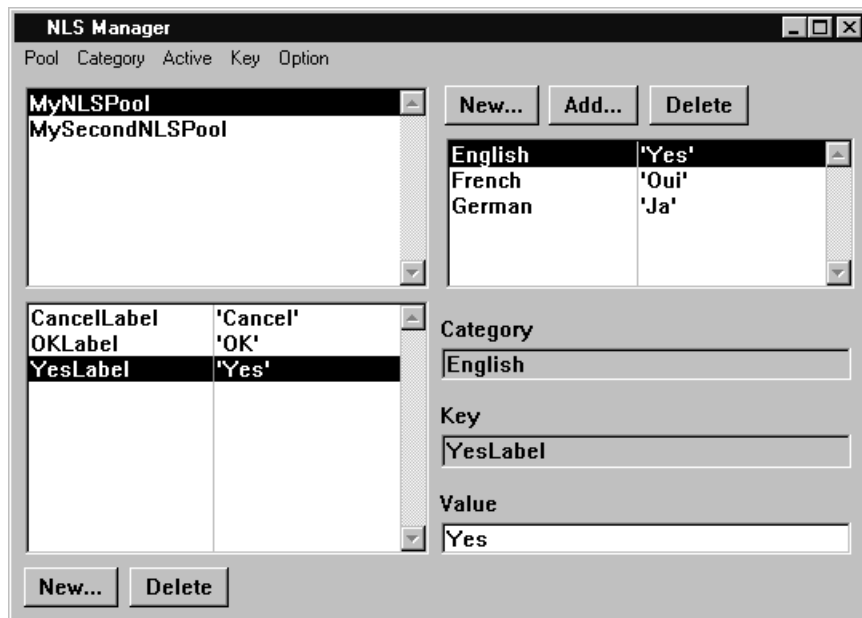
When checked, files created using the File out command will contain font definitions in portable format

### **Non Portable File Out Format**

When checked, files created using the File out command will contain font definitions in non portable format

# NLS Manager

The NLS Manager shown in Figure 13-11 is a convenient place to create, edit, store and retrieve string tables from within the VisualSmalltalk environment. The upper left, upper right and lower left panes are generic and act as described in the first section. In addition to the names, the key and category lists show the string mappings for the specified key. The edit pane in the lower right corner displays the full information on the selected key including the key and category names and the current string mapping.



**Figure 13-11** NLS Manager.

The mapping for any key and category combination may be changed by editing the contents in the Value field.

The NLS (National Language Support) manager supports two types of file: NLS files and NPL (NLS Pool file format) files. NLS files contain only key/value pairs and are in a format that can be read by Digitalk's StringDictionaryReader class. NPL files contain pool names, category names, keys, and values so that when an NPL file is loaded into the image, the pools and categories will be correctly created if they do not already exist.

Converting WindowBuilder generated windows into NLS-ized WindowBuilder generated windows is a two step process. First, use the Extract Strings command to generate NLS pool dictionaries containing strings extracted from selected



ViewManager and ApplicationCoordinator subclasses. Then use Replace Strings to replace the strings in existing ViewManager and ApplicationCoordinator subclasses with the corresponding NLS keys.

## Pool Menu

### File In

File in an NPL or NLS file. To load a NPL file programmatically (i.e., when constructing a runtime image), use the following expression:

```
WBNLSManager current load: 'NLSPOOL.NPL'
```

### File Out

File out all categories, keys, and values of the currently selected pools into an NPL format file.

### Extract Strings

This command opens the dialog shown in Figure 13-12 which displays a list of ViewManager and ApplicationCoordinator subclasses from which string will be extracted. The extraction process will gather strings from all of the specified windows and add them to the selected pool dictionary with appropriate keys.

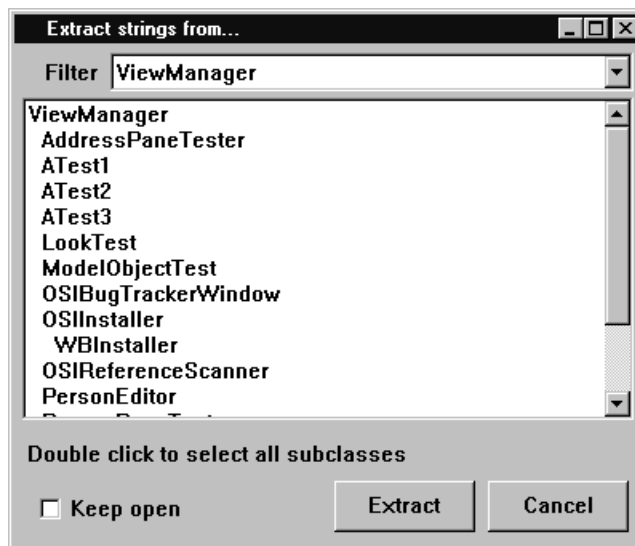


Figure 13-12 NLS Extraction Dialog.

## Replace Strings

This command opens the dialog shown in Figure 13-13 which displays a list of ViewManager and ApplicationCoordinator subclasses in which strings will be replaced with their corresponding NLS keys.

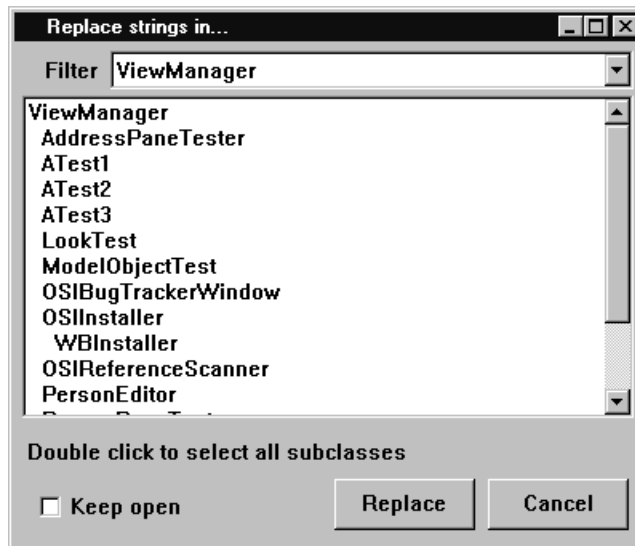


Figure 13-13 NLS Replacement Dialog.

## Category Menu

### File Out

File out the all keys and values in the selected categories and selected pools into an NPL file

### File Out Separately

File out all keys and values in each selected category and each selected pool into separate NLS files and generate a single NPL file that references all of the generated NLS files.

## Programmatic Interface

The three new pool managers, Bitmap pool manager, Font pool manager and NLS pool manager are implemented in the `WBBitmapManager`, `WBFontManager`, and `WBNLSManager` classes respectively. These three managers share the same superclass, `WBPoolManager`, and many of the same access methods. For the current pool manager, evaluate one of the following expressions:

```
WBBitmapManager current
```

```
WBFontManager current
```

```
WBNLSManager current
```

To open a pool manager window on the current pool manager, evaluate one of the following expressions:

```
WBBitmapManager current edit
```

```
WBFontManager current edit
```

```
WBNLSManager current edit
```

## Pool Manager Programmatic Assess

The following methods can be used to access the current bitmap, font, and NLS pool managers:

### Pools

**addPoolNamed:** *aPoolName*

Add a new pool with the specified name. If the pool already exists, then report an error. If the pool does not exist but a pool dictionary exists with the same name, then create a pool to manage the already existing pool dictionary. If neither the pool or the pool dictionary exist, then create both.

**load:** *fileName*

Load the .FPL, .NPL, .BPL or .BDT specified by *fileName*. Use this method to load pools into a runtime image. For example:

```
WBBitmapManager current load: 'BITMAPS.BPL'
```

**poolFor:** *aPoolRef*

Answer the pool associated with the specified pool reference or nil if there is no such pool defined in the receiver. *aPoolRef* can be a symbol, string, pool dictionary, or pool.

**poolNamed:** *aSymbol*

Answer the pool with the specified name. If such a pool does not exist, create and answer the pool using the `#addPoolNamed:` method

**pools**

Answer a collection of pools managed by the receiver

**removePool:** *aPoolRef*

Remove the specified pool from the collection of pools managed by the receiver. This does NOT delete the pool dictionary from the Smalltalk system dictionary. *aPoolRef* can be a symbol, string, pool dictionary, or pool.

## Categories

**activeCategory**

Answer the active category or nil

**activeCategory:** *aCategory*

Set the active category

**addCategory:** *aCategory*

Add a new category to each managed pool if that category does not already exist.

**categories**

Answer a collection of all categories for all managed pools

**removeCategory:** *aCategory*

Remove the specified category if it exists from each of the pools managed by the receiver. If it does not exist, then ignore it.

## Keys

**allowDuplicateKeys**

Answer true if a key in one pool is allowed to have the same name as a key in another pool.

**allowDuplicateKeys:** *aBoolean*

Set true if a key in one pool is allowed to have the same name as a key in another pool.

**includesKey:** *aKey*

Answer true if the receiver has the specified key defined in one of its managed pools

**keyAtValue:** *anObject*

Answer the key associated with the specified value. If there are multiple keys associated with the specified value, then answer the first key found. If there are no keys associated with the specified value, then report an error.

**keyAtValue:** *anObject* **ifAbsent:** *aBlock*

Answer the key associated with the specified value. If there are multiple keys associated with the specified value, then answer the first key found. If there are no keys associated with the specified value, then answer aBlock evaluated.

**keys**

Answer a collection containing all keys contained in all pool managed by the receiver.

**removeKey:** *aKey*

Remove the specified key from each of the pools managed by the receiver. If no such keys is defined, then report an error.

**removeKey:** *aKey* **ifAbsent:** *aBlock*

Remove the specified key from each of the pools managed by the receiver. If no such keys is defined, then answer aBlock evaluated.

## Access

**associationAt:** *aKey*

Answer the first association found that has the specified key as its key. If no association is found, then report an error

**associationsDo:** *aBlock*

Evaluate aBlock with each association in each managed pool

**at:** *aKey*

Answer the value corresponding the specified key in the active category. If there are multiple keys matching the specified key, then answer one of them. If there are no keys matching the specified key, then report an error.

**at:** *aKey* **ifAbsent:** *aBlock*

Answer the value corresponding the specified key in the active category. If there are multiple keys matching the specified key, then answer one of them. If there are no keys matching the specified key, then answer aBlock evaluated.

**at:** *aKey* **in:** *aCategory*

Answer the value corresponding the specified key in the specified category. If there are multiple keys matching the specified key, then answer one of them. If there are no keys matching the specified key, then report an error.

**at:** *aKey* **in:** *aCategory* **ifAbsent:** *aBlock*

Answer the value corresponding the specified key in the specified category. If there are multiple keys matching the specified key, then answer one of them. If there are no keys matching the specified key, then answer *aBlock* evaluated.

**at:** *aKey* **in:** *aCategory* **put:** *anObject*

Set the value associated with the specified key in the specified category. If the key and category do not already exist, then report an error because it is unknown in which pool the new key should be placed.

**at:** *aKey* **in:** *aCategory* **put:** *anObject* **ifAbsent:** *aBlock*

Set the value associated with the specified key in the specified category. If the key and category do not already exist, then answer *aBlock* evaluated because it is unknown in which pool the new key should be placed.

## General

**edit**

Open a pool manager window on the receiver

## Pool Programmatic Assess

Once a pool is retrieved from the current bitmap, font, or NLS pool manager, the following methods can be used to access that pool:

### Categories

**activeCategory**

Answer the active category

**activeCategory:** *aCategory*

Set the active category and copy all values associated with that category into the receiver's global pool dictionary

**addCategory:** *aCategory*

Add the specified category to the receiver if it does not already exist.

**categories**

Answer a collection containing all categories defined in the receiver

**removeCategory:** *aCategory*

Remove the specified category from the receiver if it exists.

**Keys****includesKey:** *aKey*

Answer true if the receiver's global pool dictionary contains the specified key

**keyAtValue:** *anObject*

Answer the key associated with the specified value in the active category. If such a key does not exist, then answer nil.

**keyAtValue:** *anObject* **ifAbsent:** *aBlock*

Answer the key associated with the specified value in the active category. If such a key does not exist, then answer aBlock evaluated.

**keyAtValue:** *anObject* **in:** *aCategory*

Answer the key associated with the specified value in the specified category. If such a key does not exist, then answer nil.

**keyAtValue:** *anObject* **in:** *aCategory* **ifAbsent:** *aBlock*

Answer the key associated with the specified value in the specified category. If such a key does not exist, then answer aBlock evaluated.

**keys**

Answer a collection containing all of the keys defined in the receiver's global pool dictionary

**removeKey:** *aKey*

Remove the specified key from the receiver and the receiver's global pool dictionary. If the key does not exist, then report an error.

**removeKey:** *aKey* **ifAbsent:** *aBlock*

Remove the specified key from the receiver and the receiver's global pool dictionary. If the key does not exist, then answer aBlock evaluated.

**renameKey:** *oldKey* **to:** *newKey*

Rename old key to new key in the receiver and the receiver's global pool dictionary. Use the same association object so that any compiled methods referencing that key/value pair will now reference the new key/value pair even if the method source will now be out of date.

**Access****associationAt:** *aKey*

Answer the association in the receiver's global pool dictionary with the specified key. Report an error if such an association does not exist in the receiver.

**associationAt:** *aKey* **ifAbsent:** *aBlock*

Answer the association in the receiver's global pool dictionary with the specified key. Answer *aBlock* evaluated if such an association does not exist in the receiver.

**at:** *aKey*

Answer the value in the active category associated with the specified key. If such a key does not exist, then report an error.

**at:** *aKey* **ifAbsent:** *aBlock*

Answer the value in the active category associated with the specified key. If such a key does not exist, then answer *aBlock* evaluated.

**at:** *aKey* **in:** *aCategory*

Answer the value in the specified category associated with the specified key. If such a key does not exist, then report an error.

**at:** *aKey* **in:** *aCategory* **ifAbsent:** *aBlock*

Answer the value in the specified category associated with the specified key. If such a key does not exist, then answer *aBlock* evaluated.

**at:** *aKey* **in:** *aCategory* **ifAbsentPut:** *aBlock*

Answer the value in the specified category associated with the specified key. If such a key does not exist, then store and answer the value returned by *aBlock* evaluated in the receiver at the specified key and category.

**at:** *aKey* **in:** *aCategory* **put:** *anObject*

Set the value associated with the specified key in the specified category. If the specified category is the active category then also place the value in the receiver's global pool dictionary.



## General

### **manager**

Answer the pool manager containing the receiver

### **pool**

Answer the receiver's global pool dictionary

### **poolName**

Answer the receiver's name

# Chapter 14 The Cookbook

This chapter provides a collection of recipes for getting the most out of WindowBuilder Pro and the VisualSmalltalk GUI subsystem. The chapter is broken down into sections addressing similar concepts. The major sections are:

- **Widgets**
- **Windows**
- **Menus**
- **Dialogs**
- **Potpourri**

## Widgets

### Accessing a widget programatically

Quite often, it is useful to be able to send messages to a widget while your application is running.

#### Basic Steps

1. In the Name: entryfield in WindowBuilder Pro, specify a name for the widget.
2. Send the `#paneNamed:` message to the `ViewManager`, `ApplicationCoordinator` or `CompositePane` instance with the name as the argument.

#### Example

```
showSelection  
| listWidget labelWidget |  
listWidget := self paneNamed: 'myList'.  
labelWidget := self paneNamed: 'myLable'.  
labelWidget setValue: listWidget selectedItem.
```

## Variants

If a widget is to be accessed frequently, it is advisable to assign it to an instance variable of the window. This can be accomplished by checking the checkbox next to the Name: entryfield in WindowBuilder Pro after entering a name for the widget. The name must be a valid instance variable name otherwise the checkbox will be disabled. If both of the widgets in the example above were assigned to instance variables, the example would reduce to the following:

```
showSelection  
    labelWidget setValue: listWidget selectedItem.
```

## Changing a widget's size and position programatically

Most of the time, the WindowBuilder Pro sizing and positioning facilities are quite sufficient for specifying a widget's size and position. Coupled with the Framing Editor, you can exercise precise control over how the widget resizes itself when the window is resized. Occasionally, you may need to exercise additional control over the size and position of a widget.

### Basic Steps

1. Get the widget from the window.
2. Send the #framingBlock message to the widget to return its layout specification (an instance of LayoutFrame).
3. Modify the widget's layout specification or construct a new LayoutFrame instance.
4. Send the #framingBlock: message to the widget with the new layout specification as the argument.
5. Send the #resize: message to the widget with its parent's rectangle as the argument. For widgets that are children of the main window, the parent is the window's main view.

### Example

```
moveWidgetRight  
| listWidget layoutFrame |  
listWidget := self paneNamed: 'myList'.  
layoutFrame := listWidget framingBlock.  
layoutFrame  
    leftInset: layoutFrame leftInset + 10;  
    rightInset: layoutFrame rightInset - 10.  
labelWidget  
    framingBlock: layoutFrame;  
    resize: self mainView rectangle.
```

## Changing a widget's font

Widget's will use the current system font as their default font unless you specify a different font using WindowBuilder Pro's font editor. Widget fonts may also be changed at runtime.

### Basic Steps

1. Get the widget from the window.
2. Send the `#font:` message to the widget with a new font specification as an argument.

### Example

```
makeBold
| textWidget newFont |
textWidget := self paneNamed: 'myText'.
newFont := textWidget font copy bold: true.
textWidget font: newFont.
```

## Changing a widget's color

Widget's will use their current default colors unless you specify different colors using WindowBuilder Pro's color editor. Widget colors may also be changed at runtime.

### Basic Steps

1. Get the widget from the window.
2. Send the `#backgroundColor:` or `#foreColor:` message to the widget with a new color specification.

### Example

```
setColors
| labelWidget |
labelWidget := self paneNamed: 'myLabel'.
labelWidget
  foreColor: Color blue;
  backgroundColor: Color red.
```

## Changing a widget's label or contents

Quite often, it is useful to set a static label's contents or initialize an entryfield with a value..

### Basic Steps

1. Get the widget from the window.
2. Send the `#setValue:` message to the widget with a new string as the argument.

### Example

```
setLabel
| labelWidget |
labelWidget := self paneNamed: 'myLabel'.
labelWidget setValue: 'New Label'.
```

## Hiding and showing widgets

Sometimes a widget should not be visible under certain circumstances. This can be the case with buttons that need to be hidden when their actions are not appropriate or when two alternative widgets overlap one another with only one showing at a time.

### Basic Steps

1. Get the widget from the window.
2. Send the `#hideWindow` message to the widget to make it invisible.
3. To make it visible again, send the widget the `#showWindow` message.

### Example

```
toggleWidgets
| firstWidget secondWidget |
firstWidget := self paneNamed: 'firstWidget'.
secondWidget := self paneNamed: 'secondWidget'.
firstWidget hideWindow.
secondWidget showWindow.
```

## Enabling and disabling widgets

Sometimes a widget should not be enabled under certain circumstances. This can be the case with buttons that need to be disabled when their actions are not appropriate.

### Basic Steps

1. Get the widget from the window.
2. Send the `#disable` message to the widget to make it disabled.
3. To make it enabled again, send the widget the `#enable` message.

### Examples

```
setButtonState
| button |
button := self paneNamed: 'myButton'.
self buttonState
    ifTrue: [button enable]
    ifFalse: [button disable].

executeLongOperation
| button |
button := self paneNamed: 'myButton'.
button disable.
self doLongOperation
button enable.
```

## Setting focus to a widget programatically

Sometimes it is necessary to set focus to a widget programatically. This can be the case when an invalid value is detected in an entryfield or when you want to set focus to a field in response to a user action.

### Basic Steps

1. Get the widget from the window.
2. Send the `#setFocus` message to the widget.

### Examples

```
setFocus
| entryField |
entryField := self paneNamed: 'entryField'.
entryField setFocus.
```

## Forcing a widget to redraw itself

Sometimes it is necessary to force a widget to redraw itself to update its display.

### Basic Steps

1. Get the widget from the window.
2. Send the `#redraw` message to the widget.

### Examples

```
redrawWidget  
| widget |  
widget := self paneNamed: 'widget'.  
widget redraw.
```

## Windows

### Opening a window

Quite often, it is useful to be able to open one window as the result of an action in another.

### Basic Steps

1. Send the `#open` message to a new instance of the window class.

### Example

```
openMyWindow  
  MyWindow new open.
```

### Opening a window floating above another

Quite often, it is useful to be able to open one window as a child of another window so that it floats above the first window.

### Basic Steps

1. Send the `#openWithParent:` message to a new instance of the window class with the parent window as the argument.

## Example

```
openMyFloatingWindow
    MyFloatingWindow new openWithParent: self.
```

## Opening a window as an MDI parent or child (windows only)

Quite often, it is useful to be able to open a window as an MDI parent with other windows opened as MDI children. Any window created with WindowBuilder Pro's default code generation (`#createViews`) may be opened as an MDI parent by sending it the `#openAsMDIParent` message. Any window may be opened as an MDI child by sending it the `#openWithMDIParent:` message with a single argument, the MDI parent window.

### Basic Steps

1. Send the `#openAsMDIParent` message to a new instance of a window class to create an MDI parent window
2. Send the `#openWithMDIParent:` message to a new instance of a window class with the MDI parent window as the argument to create an MDI child window.

## Example

```
openMDIWindows
    | mdiParent |
    mdiParent := MyMDIParentWindow new openAsMDIParent.
    MyMDIChild new openWithMDIParent: mdiParent.
```

If a window must always open as an MDI parent, create a `#topPaneClass` method for the window that returns **MDIFrame**:

```
topPaneClass
    ^MDIFrame.
```

## Hiding and showing windows

Sometimes a window should not be visible under certain circumstances. This can be the case when it is desirable to hide a complicated window rather than closing it and re-opening it and thus incurring the window start up costs.

### Basic Steps

1. Send the `#hideWindow` message to the window's main view to make it invisible.
2. To make it visible again, send the window's main view the `#showWindow` message.



### Example

```
closeAndCache
    self mainView hideWindow.

openFromCache
    self initializeWithNewData.
    self mainView showWindow.
```

## Bringing a window to the front of other windows

In a system with multiple modeless windows open, it is quite often desirable to bring a window to the front rather than opening another instance.

### Basic Steps

1. Send the `#bringToFront` message to the window's main view.

### Example

```
bringWindowToFront
    self mainView bringToFront.
```

## Changing a window's size and position programatically

Most of the time, the WindowBuilder Pro sizing and positioning facilities are quite sufficient for specifying a window's size and position. Occasionally, you may need to exercise additional control over the size and position of a widget.

### Basic Steps

1. Send the `#framingBlock` message to the window's main view to return its layout specification (an instance of `LayoutFrame`).
2. Modify the layout specification or construct a new `LayoutFrame` instance.
3. Send the `#framingBlock:` message to the window's main view with the new layout specification as the argument.
4. Send the `#resize:` message to the window's main view with `Display boundingBox` as the argument.

## Example

```
moveWindowRight
| layoutFrame |
layoutFrame := self mainView framingBlock.
layoutFrame
    leftInset: layoutFrame leftInset + 10;
    rightInset: layoutFrame rightInset - 10.
self mainView
    framingBlock: layoutFrame;
    resize: Display boundingBox.
```

## Changing a window's backcolor

Window's will use their current default colors unless you specify different colors using WindowBuilder Pro's color editor. Window colors may also be changed at runtime.

### Basic Steps

1. Send the #backColor: message to the window's main view with a new color.

## Example

```
setBackColor
    self mainView backColor: Color red.
```

## Enabling and disabling windows

Sometimes a window should not be enabled under certain circumstances. This can be the case when a window should be disabled during long-running operations.

### Basic Steps

1. Send the #disable message to the window's main view to make it disabled.
2. To make it enabled again, send the window's main view the #enable message.

## Example

```
executeLongOperation
    self mainView disable.
    self doLongOperation
    self mainView enable.
```

## Changing a window's label

It is often useful to change a window's label programatically to better reflect its contents.

### Basic Steps

1. Send the `#labelWithoutPrefix:` message to the window's main view with the new label as an argument.

### Example

```
setWindowLabel
    self mainView labelWithoutPrefix: 'New Window Label'.
```

## Adding a widget to a window dynamically

Sometimes you don't know in advance what widget are going to be necessary in a window. In such cases, widget may be added (and removed) to a window dynamically.

### Basic Steps

1. Send the `#addSubpaneDynamically:` message to the window's main view with the new widget as an argument.
2. Send the `#removeSubpaneDynamically:` message to the window's main view to remove a widget from the window.

### Example

```
addWidgetDynamically
    self mainView addSubpaneDynamically:
        (Button new
            setName: 'myButton';
            when: #clicked send: #doClick to: self;
            framingBlock:
                (LayoutFrame new
                    leftRatio: 0; leftInset: 10;
                    topRatio: 0; topInset: 10;
                    rightRatio: 0; rightInset: -80;
                    bottomRatio: 0; bottomInset: -50);
            yourself).
```

## Adding a timer to a window

Sometimes you need to refresh a window at a periodic time interval. It is then necessary to set up a timer and respond to the `#timer:` event.

### Basic Steps

1. Set up a handler for the window's main view's `#timer:` event.
2. Send the `#startTimer:period:` message to the window's main view. The first argument is an arbitrary Integer ID. The second argument is the period in milliseconds (one second equals 1000).
3. Stop the timer when the window closes by sending the `#stopTimer:` message to the window's main view with the previously used timer ID.

### Example

```
createViews
...
mainView
...
    when: #timer: send: #timer: to: self;
...

initWithWindow
    self mainView startTimer: 99 period: 1000.

close
    self mainView stopTimer: 99.
...

timer: id
    (self paneNamed: 'clock') setValue: Time now asString.
```

## Minimizing a window to an icon

It is often useful to minimize a window to an icon to get it out of the way.

### Basic Steps

1. Send the `#showMinimizedWindow` message to the window's main view.

### Example

```
minimizeTheWindow
    self mainView showMinimizedWindow.
```

## Restoring a window from an icon

It is often useful to restore a window from an icon if it has been minimized by the user.

### Basic Steps

1. Send the #showRestoredWindow message to the window's main view.

### Example

```
restoreTheWindow
    self mainView showRestoredWindow.
```

## Maximizing a window

It is often useful to maximize a window to make it fill the entire screen.

### Basic Steps

1. Send the #showMaximizedWindow message to the window's main view.

### Example

```
maximizeTheWindow
    self mainView showMaximizedWindow.
```

## Setting a window's icon

When a window is minimized, it is useful to associate a specific icon with it in order to identify it.

### Basic Steps

1. Send the #icon: message to the window's main view with the icon as the argument.

### Example

```
setWindowIcon
    self mainView icon: (Icon fromfile: 'myIcon.ico').
```

## Finding the window under the pointer

Occasionally, it is useful to be able to determine the window that is under the current mouse location.

### Basic Steps

1. Send the `#offset` message to the `Cursor` to get the current mouse location.
2. Send the `#windowUnderPoint:` message to the `Window` class with the mouse location to return the widget or window under the cursor.
3. Send the `#topParent` message to the object under the cursor to return the main view of the window under the cursor.
4. Send the `#owner` message to the main view to get the window itself (the `ViewManager` instance).

### Example

```
windowUnderCursor
| mousePosition underCursor mainView |
mousePosition := Cursor offset.
underCursor := Window windowUnderPoint: mousePosition.
mainView := underCursor topParent
^mainView owner
```

## Menus

### Accessing a menu programatically

Quite often, it is useful to be able to send messages to menus while your application is running.

### Basic Steps

1. Send the `#menuTitled:` message to the window's main view with the name as the menu as an argument.

### Example

```
fileMenu
^self mainView menuTitled: 'File'.
```

## Changing a menu item's label

Sometimes it is necessary to change a menu item's label to reflect the state of the application.

### Basic Steps

1. Get the menu from the window.
2. Send the `#changeItem:label:` message to the menu with the menu item's selector and new label as the arguments.

### Example

```
changeMenuLabel
| optionsMenu |
optionsMenu := self mainWindow menuTitled: 'Options'.
optionsMenu changeItem: #someOption label: 'New Label'.
```

## Enabling and disabling menu items

Sometimes a menu item should not be enabled under certain circumstances. This can be the case with menu items that need to be disabled when their actions are not appropriate.

### Basic Steps

1. Get the menu from the window.
2. Send the `#disableItem:` message to the menu with the menu item's selector as the argument to disable the menu item.
3. To make it enabled again, send the menu the `#enableItem:` message.

### Example

```
disableCutMenu
| editMenu |
editMenu := self mainWindow menuTitled: 'Edit'.
editMenu disableItem: #cut.

enableCutMenu
| editMenu |
editMenu := self mainWindow menuTitled: 'Edit'.
editMenu enableItem: #cut.
```

## Variants

To enable all of the menu items of a menu, use the `#enableAll` message. To disable all of the menu items of a menu, use the `#disableAll` message.

# Checking and unchecking menu items

Menu items may be checked and unchecked to reflect the state of the application.

## Basic Steps

1. Get the menu from the window.
2. Send the `#checkItem:` message to the menu with the menu item's selector as the argument to check the menu item.
3. To uncheck the menu item, send the menu the `#uncheckItem:` message.

## Example

```
checkOptionsMenu  
| optionMenu |  
optionMenu := self mainWindow menuTitled: 'Option'.  
optionMenu checkItem: #someOption.  
  
uncheckOptionsMenu  
| optionMenu |  
optionMenu := self mainWindow menuTitled: 'Option'.  
optionMenu uncheckItem: #someOption.
```

## Variants

To uncheck all of the menu items of a menu, use the `#uncheckAll` message.

# Removing menu items dynamically

Menu items may be deleted when no longer needed.

## Basic Steps

1. Get the menu from the window.
2. Send the `#removeItemDynamically:` message to the menu with the menu item's selector as the argument to delete the menu item.
3. To delete all of the menu items, use the `#deleteAll` message.



### Example

```
removeItemFromMenu  
| optionMenu |  
optionMenu := self mainView menuTitled: 'Option'.  
optionMenu removeItemDynamically: #someOption.  
  
removeAllItemsFromMenu  
| windowMenu |  
windowMenu := self mainView menuTitled: 'Window'.  
windowMenu deleteAll.
```

## Adding menu items dynamically

Menu items may be added dynamically if needed.

### Basic Steps

1. Get the menu from the window.
2. Send the `#insertItemDynamically:selector:atIndex:` message to the menu with the menu item's label, selector and position as the arguments.

### Example

```
addToWindowMenu: windowTitle  
| windowMenu |  
windowMenu := self mainView menuTitled: 'Window'.  
optionMenu  
insertItemDynamically: windowTitle  
selector: windowTitle  
atIndex: self numberOfWindowItems + 1.
```

## Dialogs

### Closing unwanted dialogs

At times, you may create a dialog that has no closebox, and in the process of testing it within WindowBuilder Pro, find that there is no way to remove either the dialog or WindowBuilder Pro. If this ever happens, you can access your window as follows:

```
MyDialogClassName allInstances do: [:each | each close].
```

The dialog will close, and you should find that WindowBuilder Pro is now reenabled.

## Displaying a message

Quite often, it is useful to display a simple message to the user.

### Basic Steps

1. Send the `#message: message` to the `MessageBox` class with the message text as the argument. This displays a message box titled 'Information' to the user.

### Example

```
showMessage  
    MessageBox message: 'Hi there'.
```

### Variants

To suppress the 'Information' title, use the `#messageNote: message` instead. To display a more serious warning, use the `#warning: message`. To supply your own title, use the `#notify:withText: message` instead.

## Asking a yes/no question

Frequently, an application needs to ask a user a yes or no question.

### Basic Steps

1. Send the `#confirm: message` to the `MessageBox` class with the question text as the argument. This displays a message box titled 'Please Confirm' with OK and Cancel buttons to the user.

### Example

```
askConfirmation  
    ^MessageBox confirm: 'Are you sure?'.
```

### Variants

To supply a title and use Yes, No and Cancel buttons, use the `#threeStateNotify:withText: message` instead.

## Requesting a textual response

Quite often, an application need to request a simple string from the user.

### Basic Steps

1. Send the `#prompt:default:` message to the `Prompter` class with the message text and default answer as the arguments.

### Example

```
getNewName
  ^Prompter
  prompter: 'Enter a new name:'
  default: '<new name>'.
```

### Variants

To supply a title, use the `#title:prompt:default:` method instead.

## Potpourri

### Dealing with impatient users

Often, when an application is slow to react to a mouse/keyboard event, users tend to try the action again, or they try another action before the first has completed. For instance, you might have a button called “Add” that adds an item (after some validation) from an edit field to a list. The users, seeing that nothing happens the first time they click (because it’s slow) do so again. This second click often results in a walkback.

### Possible Solutions

- Temporarily disable the button for the duration of the action.
- Temporarily unhook the `#clicked` event handler for the button.
- Temporarily disable the entire window.
- Add guard logic to your action code.
- Toss up a modal “Please Wait” dialog (add a progress indicator if you want to be really slick).
- Try using `NotificationManager>>consumeInputUntil:` (e.g., until your action completes). Try this in a test image first.

## Debugging runtime errors

Whenever you get a runtime error in something that seems to work fine during development, a red flag should go up that you may have inadvertently used some aspect of the development system (e.g., the Compiler, Debugger, Inspector, Transcript, etc.) or made a reference to a Pool Dictionary element that is not in the runtime image. For a nice, simple trick to help debug runtime apps try adding the following three lines to the beginning of the `Object>>error: method:`

```
Process copyStack
  walkbackOn: (TextWindow new openOn: '')
  maxLevels: 20.
```

Rather than a simple error window, you will get a pseudo-walkback with 20 levels (from which you should be able to identify the problem or at least the method that triggers the problem). This text in this window may then be saved out to a file. For trickier problems, you can write additional state information to the text window. As most runtime errors are of the “Message not understood” or “Runtime Error: Key is missing” variety, this minor change can be an invaluable aid in debugging this type of error.

## Fork with interrupts error

When Smalltalk is processing any of the system `#wmXXX:` messages, it disables interrupts. Dialogs may not be open when interrupts are disabled. Thus you can not open a dialog directly from any event that is handled synchronously with the OS.

Fortunately, this is pretty easy to get around. Rather than:

```
method1
  MyDialog new open
```

do this:

```
method1
  self sendInputEvent: #method2

method2
  MyDialog new open
```

The `#sendInputEvent:` message is a very useful concept to explore as it gives you a way to queue up messages for later processing. In this case, the dialog open will be queued and will execute asynchronously after the system is done processing the `#wmXXX:` message.

## Runtime-less applications

If necessary, it is possible for WindowBuilder Pro to generate code that has no dependencies on the WindowBuilder Pro runtime. This is a very useful feature for 3rd party tool providers who want to generate their interfaces but who do not want to deliver the WindowBuilder Pro runtime. It is also a useful feature for generating test cases for ParcPlace-Digitalk support that can be filed into a clean, non-WindowBuilder Pro image. The basic rule is to avoid anything that is WindowBuilder Pro specific.

Do not use:

- EnhancedEntryField, WBToolBar, WBComboBox and WBStaticGraphic
- ActionButton and LinkButton (obsolete anyway)
- Any other non-base (or non-PP-D) widgets
- CompositePanels
- FramingParameters - WindowBuilder Pro 3.1 uses LayoutFrames by default
- self paneNamed: <widgetName> - assign widgets directly to instVars instead.
- Bitmap, Font or NLS pool support. You may still NLSize your application, but you must manage the NLS pools manually.
- All WindowBuilder Pro alternative #open protocols (e.g., #openWithParent:)
- Menu items with non-alphanumeric accelerators
- #preInitWindow and #initWindow - use the TopPane #validated event instead
- WindowBuilder Pro MDI support
- Button #cancelPushButton style. Make sure to turn off the “Autorecognize OK and Cancel” property.
- DrawnButton #invisible style
- StatusPane #leftJustifiedFixed and #rightJustifiedFixed style
- Non-portable font code generation
- Optional n+1 (where n>1) event arguments (e.g., #when:send:to:withArgument:)
- Color paleGray (use Color gray instead)
- WB-PARTS support
- WBWindowDialog & WBDIALOGTopPane

## Eliminating obsolete code

The following script should help anyone who wants to track down obsolete code references in their classes. This is of primary importance to V/Win16 or V/OS2 users moving to VST 3.1. We have used this script (and variations thereof) to track down and eliminate any obsolete code references in WindowBuilder Pro in the course of porting it to VST.

```
| classes methods senders |
"Specify your classes here"
classes := (OrderedCollection new
  add: <Your Class Here>;
  add: <etc>;
  yourself).
methods := ((SmalltalkLibraryBinder contentsOf:
  (OperatingSystem isOS2
    ifTrue: ['VOBS31O']
    ifFalse: ['VOBS31W'])) at: 'methods')
collect: [:string | string
  copyFrom: (string indexOf: $>) + 2
  to: string size])
asSet asSortedCollection asArray collect: [:string |
  string asSymbol].
CursorManager execute changeFor: [
  methods do: [:aSymbol |
    senders := OrderedCollection new.
    classes do: [:class |
      senders addAll: (class sendersOf: aSymbol)].
    senders isEmpty
    ifFalse: [
      MethodBrowser new
        label: 'Obsolete Senders of: ', aSymbol;
        literal: aSymbol;
        openOn: senders]]].
```

The script will open a browser on each method in the VisualSmalltalk. obsolete SLL (you need to bind it to your image first) and show you senders of that method in whichever classes you are interested in. Many of the senders will not be problems because some of the obsolete methods share the same name as valid methods in other classes - you just need to wade through them.

You should be selective as to your target classes in order to avoid opening *lots* of method browsers and running out of memory in the process (e.g., having the system look at Object would be a bad idea). Try it with one or two of your ViewManager classes to get a feel for what it will show you.

## Migrating to the new event model

The hardest part of process in eliminating obsolete references will be in converting `#when:perform:s` to `#when:send:to:s`. We have found that to be a fairly mechanical process (and tedious). Every event in the old event model has an analog in the new event model (usually with the same name, although there are a number of exceptions that you need to be aware of). The biggest change is that the arguments that are passed to event handlers are very different than in the `#when:perform:` world.

Here are a few useful examples:

### 1) A standard button handler:

```
when: #clicked perform: #ok::
```

```
ok: aPane  
    self close
```

#### becomes:

```
when: #clicked send: #ok to: self;
```

```
ok  
    self close
```

### 2) A standard checkbox handler:

```
when: #clicked perform: #setMyBoolean::
```

```
setMyBoolean: aPane  
    myBoolean := aPane selection
```

#### becomes:

```
when: #clicked: send: #setMyBoolean: to: self;
```

```
setMyBoolean: aBoolean  
    myBoolean := aBoolean
```

### 3) A standard radiobutton handler:

```
when: #clicked perform: #setMyChoice::
```

```
setMyChoice: aPane  
    setMyChoice := self doSomething
```

#### becomes:

```
when: #turnedOn send: #setMyChoice to: self;
```

```
setMyChoice  
    setMyChoice := self doSomething
```

**4) A standard list select handler:**

```
when: #select perform: #doSelect;;

doSelect: aPane
    mySelection := aPane selectedItem
```

**becomes:**

```
when: #clicked: send: #doSelect: to: self;
doSelect: selectedItem
    mySelection := selectedItem
```

**5) A standard combobox select handler:**

```
when: #select perform: #doSelect;;

doSelect: aPane
    mySelection := aPane selectedItem
```

**becomes:**

```
when: #changed: send: #doSelect: to: self;

doSelect: selectedItem
    mySelection := selectedItem
```

**6) A standard #getContents handler:**

```
when: #getContents perform: #getListContents;;
getListContents: aPane
    aPane contents: #('A' 'B' 'C' 'D')
```

**becomes:**

```
when: #needsContents send: #getListContents:
    to: self with: aListBox;

getListContents: aPane
    aPane contents: #('A' 'B' 'C' 'D')
```

## Adapting domain models to widgets

*Courtesy of David Zeleznik - Compuserve 76547,1673*

### Problem

How does one adapt an aspect of a Domain object to a u/i widget.

### Constraints

The u/i widget is connected to one aspect of the domain object. When the user modifies the state of the widget, the underlying model's aspect must be updated. Conversely, if



the model's aspect is changed via some other independent path, the u/i widget must be refreshed to display the change in state. However, when the user interacts with the widget and updates the model, we do not want the model to circularly refresh the widget in this case.

### Applicability

An example of usage would be a text entry field that is used to both display and alter some aspect of a domain model based on its `#textChanged` event. When the model refreshes the display, there is some formatting that is performed. When the user is entering data, circular updates should be inhibited so that the user does not have the entry field reformatted while he/she is typing.

### Participants

**Model:** a domain object that has an aspect that can change states. It has `#aspect` accessor and `#aspect: mutator` methods, along with an `#aspectChanged` event.

**UIWidget:** a user interface component that is used to display the Model's aspect. It has `#stateChanged` and `#needsContents` events.

**Adapter:** this is the object that connects the Model's aspect to the UIWidget. As such, it must have a reference to each. This can be by ivar or by some form of name lookup.

### Recipe

For the sake of this recipe, we will use a `ViewManager` as an example of an Adapter. We will assume that it references the Model through a `<theModel>` ivar and references the UIWidget via name lookup. This general recipe could be factored out and used in a true Adapter hierarchy, `CompositePane`, or any other polymorphic object.

1. In the `ViewManager` create a method that refreshes the UIWidget from the Model:

```
refreshUIWidget
    (self paneNamed: 'theUIWidget') contents: theModel aspect.
```
2. Ensure that the UIWidget is refreshed when it needs updating by making `ViewManager>>refreshUIWidget` the event handler for the UIWidget's `#needsContents` event.
3. Ensure that the UIWidget is updated when the Model's aspect changes by setting an event handler for the Model's `#aspectChanged` event once the UIWidget has been created. For a `ViewManager` this can be done in `#preInitWindow`:

Technique #1 is to update the pane:

```
theModel
  when: #aspectChanged
  send: #update
  to: (self paneNamed: 'theUIWidget').
```

Technique #2 is to use the refresh method as the event handler:

```
theModel
  when: #aspectChanged
  send: #refreshUIWidget
  to: self.
```

4. Ensure that the Model is updated when the user interacts with the UIWidget by making `ViewManager>>uiWidgetChanged` the event handler for the UIWidget's `#stateChanged` event. We also prevent circular event propagation back from the Model as we are altering it.

If Technique #1 was used, above:

```
uiWidgetChanged
| theUIWidget |
theUIWidget := self paneNamed: 'theUIWidget'.
theModel
  removeActionsWithReceiver: theUIWidget
  forEvent: #aspectChanged;
  aspect: theUIWidget contents;
  when: #aspectChanged
  send: #update
  to: theUIWidget.
```

If Technique #2 was used, above:

```
uiWidgetChanged
| theUIWidget |
theUIWidget := self paneNamed: 'theUIWidget'.
theModel
  removeActionsWithReceiver: self
  forEvent: #aspectChanged;
  aspect: theUIWidget contents;
  when: #aspectChanged
  send: #refreshUIWidget
  to: self.
```

## Naming panes, event handlers and other methods

*Courtesy of Peter van Rooijen - Compuserve 100527,2375*

This discussion assumes that panes are named, not stored in ivars, and that we are working on a dialog.

## When to name a pane

Usually, you don't need to name a pane that you are not going to write to or read from in your own code.

`#createViews` may write to it, e.g. text on a button, but that is right after creation, and the button is then available in a temp var of `#createViews`.

So, usually you will not need to name pushbuttons, and static texts if they are really static. But you will need to name `entryFields`, `textPanes`, `listBoxes`. When the user has pressed OK, you will want to read their contents.

## How to name a pane

You are completely free in the names you want to give, but a systematic approach to naming will increase the quality, the speed, and the enjoyment of your work.

If the widget has a label, start its name with that. The label is what you see all the time, so it's hard to forget.

End the name with a reference to the type of output. Use `Field` for an `EntryField` (it takes a string), `List` for a `ListBox` (it takes an `OrderedCollection`), `Text` for a `TextPane` (it takes a `String(Model?)`).

## Some examples

- `firstNameField`, `middleInitialField`, `lastNameField`, `birthDateField`
- `partnersList`, `childrenList`
- `commentsText`

## How to name event handlers

Generally, you start with the name of the event, and follow that with the name of the widget that generates the event. One exception is that you use `refresh...` instead of `needsContents...`, which would sound silly. For widgets that have no name, you use the name they would have if they did.

So:

- `refreshFirstNameField`, `refreshPartnersList`, `refreshCommentsText`
- `clickedAcceptButton`, `clickedMaleButton`
- `doubleClickedPartnersList`

## Naming other methods

You may want to determine the contents of the various panes in methods whose name begins with 'contents', and is followed by the paneName. This will make their intent clear and will group them in the CHB, so they are easy to find.

What you get is:

```
contentsLastNameField
    ^self selectedPerson name lastName showString

and

refreshLastNameField
    (self paneNamed: 'lastNameField')
    contents: self contentsLastNameField.
```

To make all this #paneNamed less tedious you could define

```
ViewManager>>write: selectorSymbol to: paneNameString
    (self paneNamed: paneNameString)
    contents: (self perform: selectorSymbol).
```

So you would write

```
refreshLastNameField
    self write: #contentsLastNameField to: 'lastNameField'
```

## Tip

If you only need to set the contents of a widget when the window is opened, you don't have to provide a handler for the needsContents event, but you can simply use #write:to: in #initWindow.

When you want to display some message, don't write it out, but make a method starting with message, followed by the meaning of the message, as in #messageYouMustSelectAPartnerFirst. This way you centralize all the texts, so they are easily found and maintained.

If you define

```
ViewManager>>showMessage:messageString
    MessageBox messageNote: messageString.
```

you can write

```
messageYouMustSelectAPartnerFirst
    self showMessage: 'Please select a partner first.'
```

You can easily extend this idea to automatically set a title on the message, and choose a certain message dialog style. You could implement e.g.

ViewManager>>showNotification:aString which would pop up a dialog in the style and with the title you like to use for all notifications. When you change your preferences, there's only one method to change.

**Note:** this is an application of the general concept of code normalization, where each code element (i.e. method) is made atomic, which means that it implements only one design decision. This leads to what may be called ‘layered programming’ where factoring is pushed to the limit. It yields very compact code, which is easily maintainable.

In computer terms, code normalization trades CPU cycles for memory space: the extra indirection uses extra CPU cycles, coding everything only once saves memory space.

In human terms, it trades design effort for implementation and maintenance effort: You need extra effort at the design stage, and you save effort in implementation and maintenance.

Code normalization also automatically satisfies the single choice principle, which is explained in Bertrand Meyer’s book ‘Object Oriented Software Construction’. If you don’t have this book, get it, and study it over and over again to understand the object oriented principles for which Smalltalk is such a powerful implementation vehicle.

## Dynamic CompositePanes

*Courtesy of Herb Kelsey - Compuserve 74514,2221*

In some applications, the contents of a window might be dynamic (e.g., the user chooses which window controls are displayed on the screen at run-time). The user may also specify more controls than will fit on a single screen. It is useful to have a dynamic way of defining a large number of controls at run-time that will scroll on a screen. This can be accomplished using a ViewManager window which holds a CompositePane.

The contents of the CompositePane are defined at run-time.

### Step 1

Define a ViewManager window in WindowBuilder Pro that contains a CompositePane. The position and framing of the CompositePane will enclose all of the controls you place in the CompositePane. This works well if you don’t want to use the entire window for your dynamic controls. The CompositePane acts like a subwindow, clipping and scrolling within the region you define.

### Step 2

WindowBuilder Pro defines an `#addSubpanes` method for your subclass of CompositePane. We must replace this method. The code sample below assumes that the `propertyList` is a collection of items you want displayed on the composite pane. There are calls to `#addSubPane` where each object in the collection returns the control

it needs for a label and for editing (more on this in a minute). To the label control it sends it's name. To the edit control it sends it's current value for display in the control.

Each control that is returned is positioned vertically by the variable `verticalDU`. You could add a horizontal parameter easily. The `#widgetSpacing` method just returns the number of display units you want between controls. Once the loop on the collection is finished, the framing block for the `CompositePane` is created. This ensures that all of the controls will be drawn, not clipped. The controls will also scroll correctly if the size of the `CompositePane` in the `ViewManager` class is smaller than this value.

```
addSubpanes
| verticalDU|
verticalDU := 20.
propertyList do: [ :aProperty |
    self addSubpane: (aProperty labelWidget: self
        vertical: verticalDU name: aProperty name).
    self addSubPane: (aProperty editWidget: self
        vertical: verticalDU initialValue: aProperty value).
    verticalDU := verticalDU + self widgetSpacing +
        aProperty fullControlHeight].

"Vertical DU is now set to the extents for this
composite pane "
self
    framingBlock:
        (FramingParameters new
            iDUE: 1097 @ verticalDU;
            xC; yC; cRDU: (2 @ 2 rightBottom:
                1095 @ (verticalDU - 2)));
    backColor: Color paleGray.
```

### Step 3

The last step is to define the methods that returned the controls. There are several different properties. Below are the default methods for the subclasses that have been defined. These methods get overridden when the control has a different interface, like `ComboBoxes`, or different sizes like `TextPanels`, but this should give you enough information to see how to extend the concept.

```
fullControlHeight
    "This is the height of the control and the editWidget for
    most of the edit widgets, the larger of the label height
    and the widgetHeight."
    ^self labelHeight max: self widgetHeight

labelHeight
    ^32 "Display Units"
```

```
widgetHeight
    ^48 "Display Units"

labelWidget: owner vertical: displayUnitValue name: aString
    "Return a fully constructed Static Text control using the
    aString as the contents"
    ^StaticText new
        owner: owner;
        framingBlock:
            (FramingParameters new
                iDUE: 490 @ (self labelHeight);
                lDU: 9 r: #left; rP: 110/240;
                tDU: displayUnitValue r: #top;
                bDU: (displayUnitValue + (self labelHeight))
                r: #top);
        contents: aString;
        startGroup.

editWidget: owner vertical: displayUnitValue
initialValue: anObject
    "Return a fully constructed EnhancedEntry field. This is
    the default for most of the properties"
    ^(EnhancedEntryField new)
        owner: owner;
        framingBlock:
            (FramingParameters new
                iDUE: 530 @ (self widgetHeight);
                lP: 118/240; rDU: 23 r: #right;
                tDU: displayUnitValue r: #top;
                bDU: (displayUnitValue + self widgetHeight)
                r: #top; indent: 3 @ 4);
        startGroup;
        tabStop;
        contents: (self convertToDomain: anObject);
        yourself.
```

The main notion here is that the label and the edit control are related. The label will appear to the left of the edit control. The framing parameters are established so that the controls will grow and shrink with the movement of the composite pane. The vertical positioning is passed from the calling method. The only missing method is `#convertToDomain` which is more specific to the application. It is left in to illustrate that some care must be taken to insure that the input matches the capabilities of the control.

**Note:** Once you modify the `CompositePane #addSubpanes` method, you can no longer bring the `ViewManager` class up in `WindowBuilder Pro`. To get around this, keep the original `#addSubpanes` method and swap the two. This lets you use `WindowBuilder Pro` to clean up the main window.

# Appendix A Customizing WindowBuilder Pro

By default, WindowBuilder Pro has the ability to manipulate any widgets you create. These are added using the Add Custom Widget... command from the Add menu. After executing this command, you will be able to place, move and size any widgets you've added, and WindowBuilder Pro will generate the basic code necessary to recreate a simple copy of one of these widgets with the coordinates you specify.

This is not, however, the full extent to which WindowBuilder Pro can support customized widgets. It also provides a customizable framework you can use to augment the editing capabilities of your controls. Using this framework, you can specify how your controls can draw themselves, whether they accept a title, what their default attributes should be, and even how to edit any specific attributes you might want to set within WindowBuilder Pro.

In order to understand how this works, we must first discuss some of WindowBuilder Pro's internals, in particular the abstract class *WBGraphicObject*.

## The Graphic Object Framework

When you place a control within WindowBuilder Pro, you are not really placing a widget, but rather a lightweight representative of one called a *WBGraphicObject*. Graphic objects are WindowBuilder Pro-specific objects which internally maintain all the information associated with their corresponding real control. For each widget class you can place within WindowBuilder Pro, there is a corresponding graphic object class in the *WBGraphicObject* hierarchy.

### Graphic Object Naming

By convention, each graphic object class has the same name as its real class, preceded by the letter 'P' (for *pseudo*). For example, the graphic object class associated with the class *EntryField* is the class *PEntryField* ("pseudo entryfield").

When you place a control in WindowBuilder Pro, WindowBuilder Pro finds the matching graphic object class, creates an example instance of it, and places it within the layout pane. By creating your own graphic objects, and adding appropriate methods to



them, you can exercise extensive control over their behavior within WindowBuilder Pro.

## Setting a Widget's Contents

By default, when WindowBuilder Pro places a custom pane, the user cannot set the contents of the control; the Text: field is disabled. This can easily be changed; it's controlled by the method `usesTitle`, which by default, returns false. If you wish your control to have its contents editable within WindowBuilder Pro, add the following instance method to it:

```
usesTitle
    ^true
```

This will cause WindowBuilder Pro to a) enable the Text: field, b) keep track of the value of the Text: field, and c) generate the sending of the contents: message with the appropriate value during code generation.

## Setting a Widget's Initial Size

When a widget is placed within WindowBuilder Pro, it is created with a default size.

For example, the PButton graphic object has a suggested size based on the width and height of its label using its selected font, and when a new button is placed, it will be created with a size that surrounds its text aesthetically.

You can alter this default (or *suggested*) size for your widget by answering the message `suggestedSize`. This should answer a point representing the size in the x and y directions.

This method also controls the Autosize command within WindowBuilder Pro, with which you can select a group of controls and have them autosize themselves. WindowBuilder Pro does so by asking them each for their suggested size, and then resizing them to it.

## Setting the Minimum and Maximum Size

You may have noticed that certain controls within WindowBuilder Pro are constrained in sizing, growing and shrinking only within a limited range in the x or y direction. For example, an entryfield cannot size itself vertically. You can control this in your widget by responding to the instance messages `minSize` and `maxSize` in your graphic object. These messages should answer a point representing the appropriate pixel values of the min or max size, respectively. To illustrate, the entryfield's method for `minSize` is as follows:

```
^1@ self suggestedSize
```

In the x direction, an entryfield can get as small as one pixel; in the y direction, it's limited by its `suggestedSize` (see above).

## Working with Color

With your graphic object, there are several aspects of color editing you can control.

By default, for example, any custom controls you place can have their fore- and backcolors changed within WindowBuilder Pro. If you wish to prevent this, there are several messages you can respond to. The most restrictive message is `usesColor`; if you answer **false** to this message, the Color button will be disabled in WindowBuilder Pro. If you wish the user to specify only the backcolor of the window (for example, your widget has no forecolor), you can answer false to the message `usesForeColor`, and the Color dialog will restrict the user to choosing a backcolor only.

In addition, if you wish to change the default values for the fore- and backcolor of your widget, you can do so by responding to the messages `defaultForeColor` and `defaultBackColor`, answering an appropriate color (e.g., `Color gray`).

You can further customize your widget's color handling by overriding the supplied `colorDefaults` and `colorElements` methods. This would be useful if your widget needed to handle additional colors beyond the standard fore- and backcolors. To invoke your own color editor, override the `getColors:` method.

WindowBuilder Pro will not, by default, display with these colors in WindowBuilder Pro; you will need to do this yourself if you wish it. So what is the use of the default colors? When the user edits the colors by pressing the Color button, these will be the first values selected. Further, when code is being generated, if a color is set to the default color, WindowBuilder Pro will not generate code to set the control's color; it assumes this will already be set during initialization of the control.

## Setting the Default Font

Similarly, you can set the default font for a control. As with color, this does not mean the default font will be used when displaying text in the control; you will need to do this yourself.

## Denying Input Focus

By default, any custom panes you add will be added to the tab order list in the Tab Order Editor. If they are static controls that do not make use of the keyboard, you should add a method to your graphic object called `usesFocus`, and have it return **false**.

## Adding Styles

For many controls, there is a set of mutually exclusive styles that can be chosen to determine how the control displays/behaves (for example, pushbuttons or comboboxes). If you wish your control to have multiple styles, you can make them selectable within WindowBuilder Pro very easily.

To do so, add a method to your graphic object called `styles`. This method should answer an array of symbols, where each symbol is a message that can be sent to an instance of your control to set a particular style.

For example, you can set the style of a button by sending it the messages `defaultPushButton` or `pushButton`. The `styles` method in the `PButton` graphic object looks like the following:

```
styles
  ^#(pushButton defaultPushButton)
```

Notice that the order in which these styles are given is the order in which they'll display in WindowBuilder Pro's style combo box. Further, the first style in the list, by default, will be the default style used by WindowBuilder Pro. As with the `defaultFont` or `defaultForeColor` methods, if a graphic object's style is set to the default style, WindowBuilder Pro will not generate code for it, since it assumes this is the initial value for the control. If you want the style to always be generated, override the `generateAllStyles` method to return **true**.

If you wish to use a style other than the first one in the list, you can do so by responding to the message `defaultStyle`, answering the symbol representing the default style. Note that this style must be in the array answered by the `styles` method.

You may also set up styles that are composites of other styles. If you would like one style option to generate multiple lines of code in the widget definition, add a method called `styleMap` that returns a dictionary mapping the composite styles to arrays of symbols representing the atomic styles. This method can also be used to alias an editor style for a real style. For example:

```
styleMap
  ^super styleMap
    at: #borders put: #(#addBorderStyle);
    at: #scrollbars put: #(
      #addBorderStyle
      #addHorizontalScrollbarStyle
      #addVerticalScrollbarStyle);
    at: #verticalScrollBar put: #(
      #addBorderStyle
      #addVerticalScrollbarStyle);
  yourself.!
```

## Supporting the Layout Wizard

The widget may specify which of its events should be the default event used by the Layout Wizard in hooking it to a model by implementing a `defaultWizardEvent` method that returns the appropriate event symbol.

If the widget should have a label generated for it (like `EntryFields` or `ListBoxes`), implement a `preferredLabelOrientation` method. Returning `#left` will cause a label to be generated to the left of the widget. Returning `#top` will cause a label to be generated above the widget. Returning `nil` (the default) prevents a label from being generated.

The widget may further customize itself by responding to the `initializeForDataType:` message. See `PEnhancedEntryField` for an example.

## Enabling Morphing

If you've tried morphing one control type into another using the popup pane menu, you've noticed that `WindowBuilder Pro` provides a list of reasonable choices for you. You can have your control specify its own list and manage the transfer of special instance data that is not handled by default. There are two methods that determine the contents of the popup list: `mutationTypes` and `mutationExceptions`.

The `mutationTypes` method should return an array of Symbols that designate different control classes. Each of these classes *and their subclasses* will be included in the list. If there are certain classes that should not be included (for example, abstract classes), use the `mutationException` method to return them in a list. For `Button`, these methods are:

```
mutationTypes
    ^#(Button CPBitmapButton CPBitmapPane)

mutationExceptions
    ^#(Toggle CPHorizontalPictureButton)
```

Don't worry. Classes that don't exist in your image will simply be ignored.

If your control has its own data that it would like to initialize that is not handled by default, you can implement a `mutateSpecificsFrom:` method. This method takes the pane from which your control is mutating as its argument. Given the diverse nature of different controls, this method must be programmed very carefully. For example, if your control has an attribute called `specialValue` that it would like to grab from the original control, the method might look like this:

```
mutateSpecificsFrom: aPane
    (aPane respondsTo: #specialValue)
        ifTrue: [self specialValue: aPane specialValue].
```

The key is not to ask for something that the original control cannot provide. Depending on the nature of the data being passed, you may also have to check its type as well before using it your control (some controls use Strings for their contents, some use Collections of Strings).

## Adding Pool Dictionaries

If your widget generates code that references pool dictionaries other than `OperatingSystemConstants` or `ColorConstants`, you need to supply a `requiredPoolDictionaries` method that returns an array of pool dictionary names as symbols. WindowBuilder Pro will then automatically add those pools to the window's class definition.

## Drawing Your Widget

When WindowBuilder Pro displays the window being edited, it asks each control to display itself. This is done by sending the message `displayWith:clipRect:` to the control. You can override this method, and do your own drawing. By default in a custom control, a rectangle is drawn around it, and the control's class name is drawn in the center of the rectangle.

If you wish to provide your own drawing method, here's how it works. The `displayWith:clipRect:` method takes two arguments: the pen WindowBuilder Pro uses to draw the window and the current clipping rectangle. You perform all drawing operations with this pen, just as you would in your own control. Unlike the drawing in your custom control, the coordinates for this pen do not have their origin at the origin of the graphic object; instead, their origin is that of the layout portion of WindowBuilder Pro. If the control needs to establish custom clipping regions, it should intersect these regions with the clipping rectangle that is passed in.

To draw in terms of your graphic object's coordinate system, you will need to offset all your coordinates by a certain amount. This amount you can get from the `rect` instance variable in your graphic object, offsetting all drawing operations by `rect`'s origin. For example, the default `displayWith:clipRect:` method looks basically like the following:

```
displayWith: aPen clipRect: aRect  
    aPen rectangle: rect.
```

This also illustrates another useful feature of the `rect` instance variable: you can use it to determine the boundaries of your control as well.

In addition to the `rect` instance variable, there are other variables you can make use of to determine how to draw your control. These include:

- `font` — the current font for this control.
- `foreColor` — the current `foreColor` for this control.
- `backColor` — the current `backColor` for this control.
- `style` — the currently selected style for this control.
- `contents` — the string edited in the **Text** entryfield in WindowBuilder Pro.

In addition, there are several other convenience methods for performing the drawing of your controls; for more examples of how these can be used, you will probably want to look at the other implementors of the method `displayWith:clipRect:`; all the standard widgets are drawn in this way, and there are many examples.

If your widget draws differently under different operating systems, then you will also want to look at the `WBLookPolicy` class and its subclasses. Many of the standard widgets hand off their drawing responsibilities to the current look policy. If your widget requires this behavior, add an instance method to `WBLookPolicy` that does the default drawing. It should take the form:

```
drawMyWidget: aWidget with: aPen clipRect: clipRect
```

You may then override this method in any of the concrete `WBLookPolicy` subclasses. Your `displayWith:clipRect:` method in your widget should then look like this:

```
displayWith: aPen clipRect: clipRect
    self lookPolicy
        drawMyWidget: self
        with: aPen
        clipRect: clipRect
```

## Other Customizations

There are a few other customizations which can be useful under special circumstances.

If the widget should always be generated at the bottom of the z-order (like `GroupBoxes`), override the `staysToBack` method to return **true**.

If the widget is groupable (like `RadioButtons`), override the `isGroupable` method to return **true**.

If the widget does not work in `WindowDialogs` (basically any kind of composite control), override the `supportedInDialogs` method to return **false**.

If the widget enforces its own framing behavior (like `StatusPanels`), override the `usesFraming` method to return **false**. You may also want to override the `framingBlock:`, `configureWidget:y:width:height:borderWidth:` and `moveWidget:y:` methods. See `PStatusPane` for examples.

If the widget's visible rectangle and real rectangle are different (like a `DropDownList`), override the `hitRect` method to return the visible rectangle. This is the rectangle that is used for selecting the widget or for keeping it within its parent's boundaries when the fence is on.

When widgets are autosized, they are autosize from the left by default. You can override the `autoSizeFrom` method to specify different behavior (return `#leftJustified`, `#rightJustified` or `#centered`). See `PStaticText` for an example.

Even more esoteric customizations are possible. Careful exploration of the `WBGraphicObject` hierarchy should reveal many other possibilities to you.

## Creating an Attribute Editor

You may have noticed that you can edit other attributes of some controls in WindowBuilder Pro besides the basic events, contents, color, etc., using the Attribute editor. For example, you can set the minimum, maximum, line increment, and page increment of a scrollbar in this manner. This is a very powerful facet of WindowBuilder Pro customization; using it, you can create editors to alter any facet of your control, using whatever user interface you wish. To make this easy for you, `WBGraphicObject` provides a framework with which you can edit, transfer, and generate code for any other properties you might be interested in editing.

There are four WindowBuilder Pro-specific tasks you will need to deal with in order to edit your own attributes:

- copy the attributes from a real widget to your graphic object
- copy the attributes from one graphic object to another
- provide an editor that can set the attributes
- generate the code that represents the attributes

We will address each of these processes in turn.

First we need to copy the attributes out of a real widget into your graphic object. To see why, consider how WindowBuilder Pro is able to re-edit a window definition: it first creates a real window, containing real widgets. It converts this window into graphic objects, by asking each widget in the window in turn for information about itself. It then discards the real window, and displays the graphic objects. Any attributes you wish to be editable by WindowBuilder Pro will need to be transferred from the real widget to the graphic object during this process.

To do so, you will need to create a method called `readSpecificsFrom:` in your graphic object. This method receives one argument: the real widget that is being

converted. You will need to copy any attributes out of the real widget passed in into your graphic object. To illustrate, here is the method used in the PScrollBar class:

```
readSpecificsFrom: widget
self
    minimum: widget minimum;
    maximum: widget maximum;
    lineIncrement: widget lineIncrement;
    pageIncrement: widget pageIncrement.
```

Note that the accessor methods in the PScrollBar class had to be created; these methods simply set instance variables. Note also that the real widget required accessor methods as well.

The next task we need to accomplish is copying the attributes from one graphic object to another. This is necessary because WindowBuilder Pro provides the ability to copy and paste controls, and needs a way to transfer all the information about a control from one graphic object to another.

Copying between graphic objects is very similar to copying from a real widget. You need to respond to the method `copySpecificsTo:` (which passes in the new graphic object as an argument), and copy all the attributes of `self` into this new graphic object. As an example, here is the `copySpecificsTo:` method used by PScrollBar:

```
copySpecificsTo: aPane
aPane
    minimum: self minimum;
    maximum: self maximum;
    lineIncrement: self lineIncrement;
    pageIncrement: self pageIncrement.
```

As with the `readSpecificsFrom:` method, you will need to make sure accessor methods (and associated instance variables) are provided in your graphic object class.

The next task we need to accomplish is actually generating the code necessary to reproduce the attributes the user has specified. This is actually very easy to do, as long as the attributes are easily representable in code.

As WindowBuilder Pro generates code, it passes a stream to each of the graphic objects within it, asking them each to store their own code on the stream.

To respond to this, you will need to add the method `storeSpecificsOn:indentString:` to your graphic object class. The first argument to this method is the stream WindowBuilder Pro is generating code on; the second is a string of whitespace used to provide indentation. To illustrate a good use of this mechanism, let's look at a fragment of the method used by PScrollBar to generate its code:

```
storeSpecificsOn: aStream indentString: indentString
    self minimum = 0
```



```
ifFalse: [  
    aStream  
        nextPutAll: ';; cr;  
        nextPutAll: indentString;  
        nextPutAll: 'minimum: ', self minimum asString].  
...
```

This method illustrates several things. First, note that it does not generate code if the minimum value is 0; this is a way of minimizing code generation, since it assumed that the default minimum value in a widget is 0. Next, note that the first code generated is a semicolon (;) followed by a carriage return. This terminates the previous line of code, and is the way in which you should generate code as well. Finally, note the method `asString` that is sent to `self minimum`; this is a message you can send to any object that will display itself as a string — we needed to send this here since a stream would not be able to print an integer value. The `asString` method is provided for most objects; you can override it if you wish.

Finally, now that we're able to read in our attributes, copy them, and generate code for them, it would be nice to actually provide an editor for changing them. To do so, we need to create one, and provide a way to utilize it.

Creating an attribute editor is fairly straightforward. First, create a new dialog with WindowBuilder Pro, containing the controls you will need to edit this pane. Note that it must be a subclass of the ViewManager class `WBAttributeEditor` (hold down the ALT key when saving the window), which provides the machinery necessary to interface with WindowBuilder Pro; you can specify the superclass when you save the interface.

Once you've generated the code, you'll need to do a little manual coding to handle the input and output of the dialog.

First, you will want to create an `initWindow` method to initialize the values of any widgets in your editor, using the instance variable `thePane`. This variable will have already been set up for you, and contains the graphic object that is currently being edited. As an example of this initialization, the scrollbar editor class `WBScrollBarEditor`, contains the following initialization method:

```
initWindow  
    (self paneNamed: 'lineInc')  
        contents: thePane lineIncrement asString.  
    (self paneNamed: 'pageInc')  
        contents: thePane pageIncrement asString.  
    (self paneNamed: 'minimum')  
        contents: thePane minimum asString.  
    (self paneNamed: 'maximum')  
        contents: thePane maximum asString.
```

Next, you will need to add methods for dealing with closing the dialog. Typically, a dialog has an OK button and a Cancel button. By default, the `WBAttributeEditor` class

already provides the appropriate `cancel` method, so if you add a Cancel button, you need only have it send the message `cancel` when clicked. To deal with the OK button, we need to create a method, `ok`, and associate it in WindowBuilder Pro with the clicked event of the OK button.

In the `ok` method, we need to update the attributes in `thePane`, to reflect the changes in the controls the user has changed. Here is the `ok` method for `WBScrollBarEditor`

```
ok
    thePane
        lineIncrement: (self paneNamed: 'lineInc')
            contents asInteger;
        pageIncrement: (self paneNamed: 'pageInc')
            contents asInteger;
        minimum: (self paneNamed: 'minimum')
            contents asInteger;
        maximum: (self paneNamed: 'maximum')
            contents asInteger.
```

If you wish your dialog to be more interactive, you can do so; just make sure `thePane` is not changed unless the user explicitly says `ok` — otherwise, you will be changing the object that is being edited in WindowBuilder Pro.

That's all there is to it; once you've completed these four steps, WindowBuilder Pro will be able to edit the specific attributes you're interested in.

In certain cases, using the attribute editor should force either the control's text or size to change. If this is true you can tell WindowBuilder Pro to make the changes when it returns. Implementing a `changesText` method that returns **true** will cause the control's text to update (`StaticText` uses this). Implementing a `changesSize` method that returns **true** will cause the control to automatically resize itself (`WBToolBar` uses this).

Once your attribute editor is complete, you need to hook it to your pseudo widget class by adding an `attributeEditor` method that returns your new attribute editor class.

## Pre and Post Attribute Editing

If any special pre- or post-editing operations are required, you may override either the `preEdit:` or `postEdit:` methods in your widget definition. Both methods should return **true** if they were successful and **false** if they were not. Returning **false** from the `preEdit:` method will prevent the attribute editor from even opening.

## Adding Widget Palette Icons

Once you have created your control wrapper, WindowBuilder Pro provides an easy mechanism to gain access to it from within the editor. While one can always use the Add Custom Widget command to add your control to the Custom Widget list, it more useful (and more satisfying) to add it directly to one of WindowBuilder Pro's widget palettes (or to one of your own!).

Here are the steps you need to follow to add your control to a widget palette:

1. Bring up the System Bitmap Manager (hold down the CTRL key when launching the Bitmap Manager).
2. Create a 28@28 button to represent your control. You can start by duplicating one of the existing control buttons, or by creating a new one. The Button Edit command can make this much easier. A widget palette button is represented as a single bitmap twice as wide as the button itself. The left half of the bitmap contains the "up" version of the button while the right side contains the "down" version.
3. Name your button the name as your control class plus 'SysBitmap' (e.g., 'MyControlSysBitmap').
4. Create a simple *Add-In* class for use with the Add-In Manager may use any of the existing Add-Ins as a template (they are all `WBAbstractAddInModule` subclasses).

Use the Add-In Manager to enable your Add-In.

## 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:

- **commonName** - Answer the name of the add-in.
- **comment** - Answer a description of the add-in.
- **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 multi-view ViewManagers in WindowBuilder Pro.

```
WBAbstractAddInModule subclass: #WBMultiViewAddInModule
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

!WBMultiViewAddInModule class methods !
comment
    ^'WARNING!! Using multiple views
    is discouraged and may not
    be supported in future releases'!

commonName
    ^'Multiple Views Support'!

modifiesMenus
    "Does this add-in modify the menus?"
    ^true!
```

```
modifyMenus: aMenu
    "Modify the menus"
    self
        insertMenu: #('\uView' nil
            ((('\uSwitch To...' menuViewSwitchTo ''
                hasMultipleViews nil nil)
                ('\uCreate...' menuViewCreate '' nil nil nil)
                ('\uRemove...' menuViewRemove ''
                    hasMultipleViews nil nil)))
        in: aMenu
        after: 'Edit'!!
```

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).

Menus are defined as an array of three values: the menu label, the menu selector, and the menu accelerator key. For nested menus, the third slot is replaced by an array of arrays defining submenus. Menus may cascade up to one level. *Multiple cascade levels are not supported.*

The second example illustrates adding new widgets to the tool palette and the Add menu:

```
WBAbstractAddInModule subclass: #WBWindows95AddInModule
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: '' !

!WBWindows95AddInModule class methods !
comment
    ^'Adds the Windows 95 widgets
    to the WBPro palette. Warning:
    Win95 widgets may or may not
    be portable to OS/2'!

commonName
    ^'Windows 95 Widgets'!

modifiesPalette
    "Does this add-in modify the palette?"
    ^true!
```

```

modifyPalette: thePalette
    "Modify the palette"
    self
        insertMenu: #('\uVideoPane' 'VideoPane' '')
            in: thePalette
            after: 'StatusPane';
        insertMenu: #('Windows 95' addWidget:
            (
                ('\uHeader' 'Header' '')
                ('\uListView' 'ListView' '')
                ('\uProgressBar' 'ProgressBar' '')
                ('\uRichEdit' 'RichEdit' '')
                ('\uStatusWindow' 'StatusWindow' '')
                ('\uTabControl' 'TabControl' '')
                ('Trac\ukBar' 'TrackBar' '')
                ('Tree\uView' 'TreeView' '')
                ('\uUpDown' 'UpDown' '')
            ))
        in: thePalette
        after: 'Composite'!!

```

The example notifies the system that it is interested in modifying the palette (which also modifies the Add menu) and does so via the #modifyPalette: method.

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 !
commonName
    "Answer the name of the Add-In"
    ^`Sample Add-In #2`!

comment
    "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);
            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. Two properties 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**

Answer the category in which the method should be included. The default is “window definition”.

**defaultMessageArguments**

Answer the collection of method arguments. For each key word in the generated method, there should be one argument provided.

**defaultMessageSelector**

Answer the method name (as a `Symbol`) to be generated. All of the `WBCodeModule` classes override this method.

**initializeCodeGeneration**

Perform special initializations before the method is generated.

**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.

**generateComment**

Generate the comment body for the method. By default, this will generate the comment “Private”.

**generateTemporaries**

Generate the temporaries for the method.

**stream**

Answer the value of the stream instance variable. This is the current stream that the method is being generated to.



**targetClass**

Answer the class in which to create the method. This should be the class of the object for which code is being generated.

**targetObject**

Answer the object for which code is being generated.

You can now generate code using an expression similar to the following:

```
WBCodeGenerator generateCodeUsing:  
    (SomeCodeModule new  
        object: <object to generate>;  
        targetClass: <class to generate code into>)
```

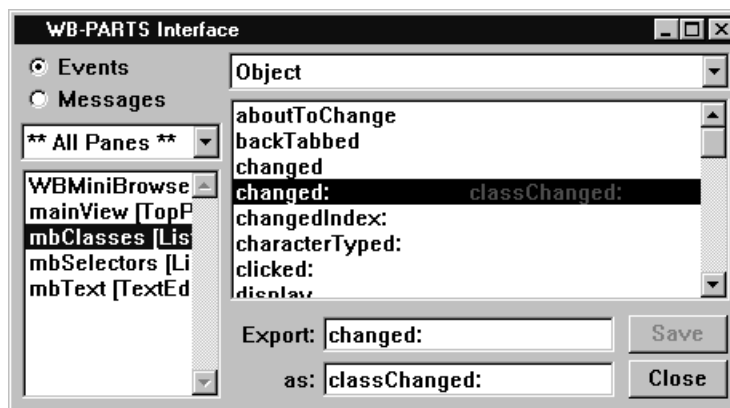
If you want you additional code module executed every time WindowBuilder Pro generates code, create an Add-In module that returns true for `modifiesCodeGeneration` and adds your new module via `modifyCodeGeneration:`.

## Appendix B PARTS Workbench Integration

Windows generated by WindowBuilder Pro can be used in any PARTS application. After building the window using WindowBuilder Pro, open the WB-PARTS Interface window to define which events and messages are visible to the PARTS programmer. Once the PARTS interface is defined, you may drag the PARTS icon in the WindowBuilder layout window and drop it in the PARTS Workbench.

### WB-PARTS Interface

To open the WB-PARTS Interface window, click on the PARTS icon in the WindowBuilder layout window. For the WBMiniBrowserExample (see source code at the end of this appendix), the WB-PARTS Interface window would look something like the following:



**Figure B-1** WB-PARTS Interface editor.

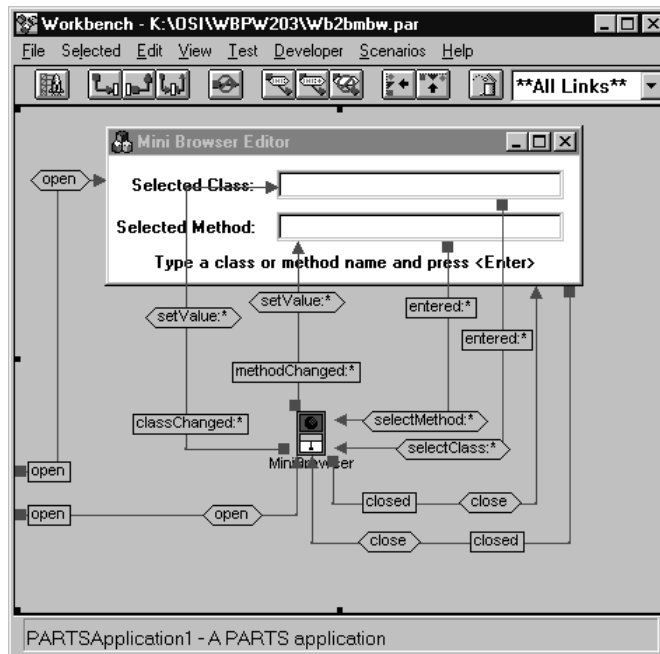
The lower left pane in the WB-PARTS Interface window contains a list of names of all widgets that you have defined. The drop down list above it contains filters for this list. You may show all widgets or only those widgets of a particular class.

Selecting a widget name displays a series of events or messages in the right hand pane. The drop down list above it contains filters for this list. You may show all events or messages, or only those events or messages defined in a particular class.

Each of these events or messages may be exported as part of the WB-PARTS interface for your window. Selecting an event or message in the right hand pane displays the original name of the event or message in the Export: entryfield. To specify that an event or message be exported, type the new name of the event or message in the As: entryfield. This new name will be used whenever linking events or messages in the PARTS Workbench.

In the WBMiniBrowserExample, the `changed:` event from the upper left list box is exported as the `classChanged:` event. This is accomplished by selecting the class list box in the left most list box, selecting `changed:` in the right most list box, then typing the new event name (`classChanged:`) in the text edit pane in the lower right corner of the WB-PARTS Interface window. Once this is defined in the WB-PARTS Interface window, whenever the upper left list box selection changes, the `classChanged:` event is triggered.

Once the WB-PARTS Interface is defined, drag the parts icon from the WindowBuilder Pro window and drop it in the PARTS Workbench. Your WindowBuilder Pro defined window will appear as a single icon in the PARTS Workbench. You may link events and messages to this icon in the normal PARTS fashion. For example, in the WBMiniBrowserExample, the PARTS Workbench looks like this:



**Figure B-2** PARTS Workbench integration example.

# WBMiniBrowserExample code

```

ViewManager subclass: #WBMiniBrowserExample
  instanceVariableNames:
    ' selectedClass selectedMethodSelector '
  classVariableNames: ''
  poolDictionaries:
    ' ColorConstants OperatingSystemConstants ' !

!WBMiniBrowserExample class methods !
wbBasicVersion
  "Private - Answer the WindowBuilder Pro version.
  Generated in: VisualSmalltalk Enterprise 3.1.0 Win32"

  ^3.1!

wbPartEventItems
  "WARNING!! This method was automatically generated by
  WindowBuilder. Code you add here which does not conform to
  the WindowBuilder API will probably be lost the next time
  you save your wb-parts definition."

  #generated.
  ^#(classChanged: closed methodChanged: textChanged:)! !

!WBMiniBrowserExample methods !
classSelected: selectedItem
  "Callback for the #changed: event triggered in the
  ListBox named ''.
  (Generated by WindowBuilder)"

  selectedClass := selectedItem notNil
    ifTrue: [ Smalltalk at: selectedItem asSymbol ]
    ifFalse: [ nil ].
  selectedMethodSelector := nil.
  self updateSelectors.
  self updateText.!

classSelectors
  "Answer the method selectors for the selected class."
  ^selectedClass selectors asSortedCollection asArray
    collect: [ :each | each asString ]!

```

```
classText
    "Answer the description for the selected class."
    | stream |
    stream := WriteStream on: (String new: 100).
    selectedClass fileOutOn: stream.
    ^stream contents.

createViews
    "WARNING!! This method was automatically generated by
    WindowBuilder. Code you add here which does not conform to
    the WindowBuilder API will probably be lost the next time
    you save your layout definition."

    | mainView mbClasses mbSelectors mbText xDU yDU |

    #generated.
    xDU := WindowDialog dialogUnit x / WindowDialog
    unitMultiplier x.
    yDU := WindowDialog dialogUnit y / WindowDialog
    unitMultiplier y.
    mainView := self topPaneClass new.

    "Temporary Variables"
    mbClasses := ListBox new.
    mbSelectors := ListBox new.
    mbText := TextEdit new.

    mainView
        owner: self;
        setName: 'mainView';
        labelWithoutPrefix: 'Mini Browser';
        noSmalltalkMenuBar;
        backColor: Color gray;
        framingBlock: (
            LayoutFrame new
                leftRatio: 1/2; leftInset: -78 * xDU;
                topRatio: 1/2; topInset: -121/2 * yDU;
                rightRatio: 1/2; rightInset: -78 * xDU;
                bottomRatio: 1/2; bottomInset: -103/2 * yDU).
    self addView: mainView.

    mbClasses "ListBox"
        owner: self;
        setName: 'mbClasses';
        when: #changed: send: #classSelected: to: self;
```

```

    framingBlock: (
        LayoutFrame new
            leftRatio: 0; leftInset: 0 * xDU;
            topRatio: 0; topInset: 0 * yDU;
            rightRatio: 36/73; rightInset: 0 * xDU;
            bottomRatio: 84/173; bottomInset: 0 * yDU);
    startGroup;
    font: SysFont;
    contents: #( 'Array' 'Dictionary' 'OrderedCollection'
        'Point' 'Rectangle' 'Set' ).
mainView
    addSubpane: mbClasses;
    subPaneWithFocus: mbClasses.

mbSelectors "ListBox"
    owner: self;
    setName: 'mbSelectors';
    when: #changed: send: #methodSelected: to: self;
    framingBlock: (
        LayoutFrame new
            leftRatio: 36/73; leftInset: 0 * xDU;
            topRatio: 0; topInset: 0 * yDU;
            rightRatio: 1; rightInset: 0 * xDU;
            bottomRatio: 84/173; bottomInset: 0 * yDU);
    startGroup;
    font: SysFont.
mainView addSubpane: mbSelectors.

mbText "TextEdit"
    owner: self;
    setName: 'mbText';
    framingBlock: (
        LayoutFrame new
            leftRatio: 0; leftInset: 0 * xDU;
            topRatio: 84/173; topInset: 0 * yDU;
            rightRatio: 1; rightInset: 0 * xDU;
            bottomRatio: 1; bottomInset: 0 * yDU);
    addHorizontalScrollbarStyle;
    addVerticalScrollbarStyle;
    startGroup;
    noTabStop;
    font: SysFont.
mainView addSubpane: mbText!

```

```
methodSelected: selectedItem
    "Callback for the #changed: event triggered in the
    ListBox named ' '.
    (Generated by WindowBuilder)"

    selectedMethodSelector := selectedItem notNil
        ifTrue: [ selectedItem asSymbol ]
        ifFalse: [ nil ].
    self updateText.

methodText
    "Answer the selected method source"
    ^(selectedClass
        compiledMethodAt: selectedMethodSelector
        ) source!

updateSelectors
    "Update the method selectors list pane."
    (self paneNamed: 'mbSelectors')
        contents: (
            selectedClass isNil
                ifTrue: [ #( ) ]
                ifFalse: [ self classSelectors ] );
    triggerChanged.

updateText
    "Update the text pane."
    (self paneNamed: 'mbText') contents: (
        selectedClass isNil ifTrue: [ ' ' ] ifFalse: [
            selectedMethodSelector isNil
                ifTrue: [ self classText ]
                ifFalse: [ self methodText ] ] ).!

wbPartEvents
    "WARNING!! This method was automatically generated by
    WindowBuilder. Code you add here which does not conform to
    the WindowBuilder API will probably be lost the next time
    you save your wb-parts definition."

    #generated.
    ^WBPARTSEventList new
        items: self class wbPartEventItems
        separators: #( )
        defaultItem: #classChanged;;
```

```

addWbItem: (
    WbPartEvent
        sourceName: 'mbClasses'
        sourceSelector: #changed:
        receiver: self
        triggeredEvent: #classChanged: );
addWbItem: (
    WbPartEvent
        sourceName: 'mainView'
        sourceSelector: #closed
        receiver: self
        triggeredEvent: #closed );
addWbItem: (
    WbPartEvent
        sourceName: 'mbSelectors'
        sourceSelector: #changed:
        receiver: self
        triggeredEvent: #methodChanged: );
addWbItem: (
    WbPartEvent
        sourceName: 'mbText'
        sourceSelector: #changed:
        receiver: self
        triggeredEvent: #textChanged: );
yourself!

```

wbPartMessages

"WARNING!! This method was automatically generated by WindowBuilder. Code you add here which does not conform to the WindowBuilder API will probably be lost the next time you save your wb-parts definition."

#generated.

```

^WBPARTSMessageList new
    items: #(close inspect open selectClass:
        selectMethod: text:)
    separators: #( )
    defaultItem: #open;
addWbItem: (
    WbPartMessage
        source: self
        sourceSelector: #selectClass:
        receiverName: 'mbClasses'
        selector: #selectItemAndTriggerChanged: );

```



```
addWbItem: (
    WbPartMessage
    source: self
    sourceSelector: #selectMethod:
    receiverName: 'mbSelectors'
    selector: #selectItemAndTriggerChanged: );
addWbItem: (
    WbPartMessage
    source: self
    sourceSelector: #text:
    receiverName: 'mbText'
    selector: #contents: );
yourself! !
```

---

# Appendix C WindowBuilder Pro Tools Menu

The **Tools and Inspectors** Add-In adds a Tools menu to the WindowBuilder Pro menu bar. This menu contains a number of useful options for inspecting and manipulating WindowBuilder Pro elements and for inspecting a number of standard system objects. This appendix describes the various options that are available on this menu.

## Inspect Selections

Open an inspector on the currently selected pseudo widgets or the window if no widgets are selected.

## Inspect Other

Displays a submenu with the following choices:

### Self

Open an inspector on WindowBuilder Pro itself.

### Prototype

Open an inspector on the prototype object managed by WindowBuilder Pro. The prototype holds onto various pieces of information concerning the currently edited window.

### All Instances

Open an inspector on all instances of the currently edited window class.

### Scrolling Form

Open an inspector on WindowBuilder Pro's scrolling form.

### Layout Form

Open an inspector on WindowBuilder Pro's layout form.

### **Children**

Open an inspector on all of the widgets in the currently edited window.

### **Menu Bar**

Open an inspector on the pseudo menubar of the currently edited window.

### **Clipboard**

Open an inspector on WindowBuilder Pro's clipboard.

### **Smalltalk**

Open an inspector on the Smalltalk global dictionary.

### **OperatingSystemConstants**

Open an inspector on the OperatingSystemConstants pool dictionary.

### **ColorConstants**

Open an inspector on the ColorConstants pool dictionary.

### **CharacterConstants**

Open an inspector on the CharacterConstants pool dictionary.

### **CursorConstants**

Open an inspector on the CursorConstants pool dictionary.

### **GraphicsConstants**

Open an inspector on the GraphicsConstants pool dictionary.

### **SystemColorConstants**

Open an inspector on the SystemColorConstants pool dictionary.

### **SystemValueConstants**

Open an inspector on the SystemValueConstants pool dictionary.

### **VirtualKeyConstants**

Open an inspector on the VirtualKeyConstants pool dictionary.

## **Browse**

Displays a submenu with the following choices:

### **Classes**

Open an class hierarchy browser on all classes.

### **Packages**

Open an package browser.

### **Services**

Open the Services Manager.

### **Window**

Open an class hierarchy browser on the Window hierarchy.

### **SubPane**

Open an class hierarchy browser on the SubPane hierarchy.

### **ViewManager**

Open an class hierarchy browser on the ViewManager hierarchy.

### **ApplicationCoordinator**

Open an class hierarchy browser on the ApplicationCoordinator hierarchy.

### **Event Manager**

Open an class hierarchy browser on the EventManager hierarchy.

## **Outboards**

Displays a submenu with the following choices:

### **Inspect**

Open an inspector on all of WindowBuilder Pro's outboard windows.

### **Close All**

Close all of WindowBuilder Pro's outboard windows.

## **Undo Manager**

Displays a submenu with the following choices:

### **Inspect**

Open an inspector on all of WindowBuilder Pro's undo manager.

### **Clear Undos**

Clear out all of the undo/redo records that have built up.

## **Code Generation**

Displays a submenu with the following choices:

### **Generate Test Code For Selection**

Generate test code for the currently selected widgets.

### **Debug...**

Step through the WindowBuilder Pro code generation process for the current window. This command brings up a walkback window on purpose.

## **Initialize**

Displays a submenu with the following choices:

### **Attributes**

Initialize all of the widget attributes.

### **Extras**

Initialize all of the Add-Ins.

### **Properties**

Initialize the WindowBuilder Pro properties back to their original defaults.

## **Other**

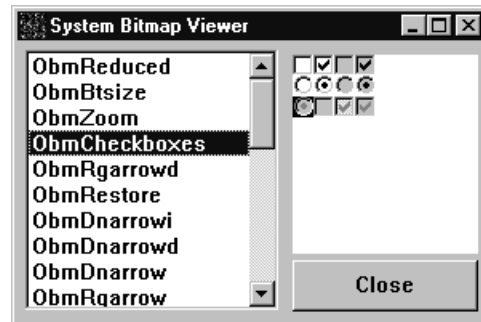
Displays a submenu with the following choices:

## Fix Framing

Little used command for fixing the framing on dialogs imported from Smalltalk/V OS/2 2.0 that use proportional top and bottom framing specifications. Only use this command if you have loaded an old window definition into WindowBuilder Pro and some of the widgets look up-side-down.

## View System Bitmaps

Launches a simple viewer for looking at the built in operating system bitmaps. Shown below:



## Dirty

Set the WindowBuilder Pro changed flag to true. This will enable you to make WindowBuilder Pro believe that a window has changed, thus allowing you to issue the save command and regenerate the code.



## Appendix D The Notifier Explained

The Notifier is one of the most difficult aspects of Smalltalk/V to understand. Unfortunately, it is often necessary to understand the Notifier in order to understand how operating system events get translated into their Smalltalk counterparts. The following discussion is intended for advanced Smalltalk users and is excerpted from Scott Wlaschin's forthcoming *Advanced Smalltalk/V* book. For more details on the book contact:

Scott Wlaschin  
Radical Systems  
Los Angeles, CA 90068  
Compuserve: 71441,2442

### What is the Notifier?

The global object Notifier is an instance of NotificationManager. It is responsible for:

1. Handling input events from the operating system
2. Converting the event into an appropriate selector
3. Finding the window that event is meant for, and directing the message to that window.

### Programming for OS/2 and Windows

Before we get into the details of the Notifier, it is helpful to have a quick overview of how OS/2 and Windows applications must be written.

#### Window Events

GUIs are *event driven*, that is, the application does not initiate actions, it responds to events. Mostly, the events come from the user pressing the keyboard or clicking the mouse. Some events are generated by the system, such as timer events, and some events are sent by other objects in the system.

In OS/2 and Windows, a 'window' is the basis of GUI programming. Windows are used to implement a crude kind of object-orientation:



They receive messages from the user.

- They receive messages from other windows.
- They can send messages to other windows.
- Different windows can respond differently to the same message (polymorphism).

Each window has a *window handle*, which identifies it uniquely in the system. Events are generally sent to a particular window, and are tagged with the corresponding window handle.

## The Message Loop

Every application has an *application queue*, which stacks messages sent to the application. It is the responsibility of the application to fetch events from the queue and process them appropriately. To do this, the application must have a *message loop* that looks something like this:

```
while (GetMessage (&msg, NULL, NULL, NULL)) {  
    /* do some minimal pre-processing */  
    TranslateMessage(&msg);  
  
    /* send the message to the appropriate window */  
    DispatchMessage(&msg);  
}
```

GetMessage fetches a message from the application queue. You can then do some pre-processing of the message (TranslateMessage) before sending the message to the appropriate window (DispatchMessage).

The main purpose of the message loop is to return control to the operating system frequently (in GetMessage). This is needed because each application has to share the desktop with other applications, and should not hog the interface. Even in a true pre-emptive multitasking system such as PM for OS/2, the user interface for all applications is handled in a serial fashion.

## The Window Function

When each window is created, a window function must be specified. This is the function that will determine how to handle the events that are sent to it.

Each window can have a different window function. If a window loosely corresponds to an object, then the window function loosely corresponds to the method dictionary. Each window can have a different window function, and windows which share the same window function are said to have the same class (I wonder where they got that from!).

The window function has four parameters:

- The window handle.
- The message.
- Two extra parameters (used as parameters to the message).

A simplified window function might look like this:

```
MyWindowProc( hWnd, message, wParam, lParam) {
    switch (message) {

        case WM_CHAR:
            ...handle a keystroke...

        case WM_LBUTTONDOWN:
            ...handle a left button click...

        default:
            ...call the default window function to
            handle the message...

    } /* end switch */
} /* end function */
```

## An Overview of Event Processing in Smalltalk

Smalltalk was designed as a virtual machine, and expects to ‘own’ the entire system. One of the great achievements of Digitalk has been to smoothly integrate the Smalltalk/V environment with the host environment, with minimal disruption to the internal workings of the virtual machine.

Smalltalk also has a kind of message loop, which is a superset of the basic message loop that is required for all applications. Since the word ‘message’ is misleading in a Smalltalk context, I will call the message loop the *event loop* instead.

The **Notifier** is responsible for executing the event loop, in the method `run`.

There are three steps in the loop.

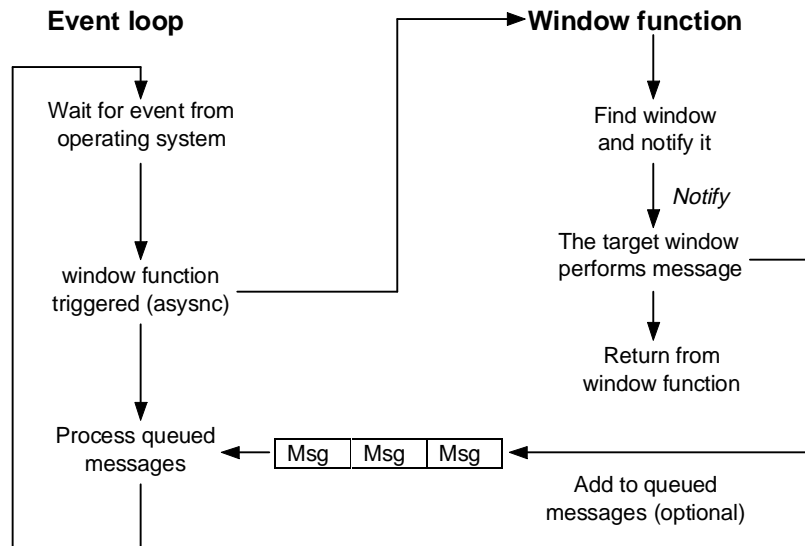
1. A event is fetched from the application queue. If there is no event, control is yielded to the operating system.
2. When an event is available, the window function for Smalltalk/V is triggered (asynchronously). The window function is hidden in the virtual machine, but it immediately calls a Smalltalk method to notify the window.

Notification is the task of

1. Finding the window to which the message was sent.
2. Converting an integer message number into a Smalltalk selector.
3. Asking the window to perform the message.

Typically, a window will add the message to a global message queue, and return immediately. How the individual events are actually processed by the window is discussed later in this section.

3. Finally, all messages in the global message queue are handled.



The main event loop for Smalltalk.

## Executing the Event Loop

The main event loop is in the method `#run`.

```
NotificationManager>>run
```

This method performs the event loop as described above. It does not contain the notification methods. They are triggered separately by the window function.

`#run` is called when Smalltalk starts up. It must also be called whenever a new process becomes the user interface process.

## Step 1: Wait for an Event

The application queue is ‘peeked’ to see if a message is waiting. If there is no message, control is yielded to the operating system.

The method used differs between Windows and OS/2. In Windows, the method is `#readWinQueue`, in OS/2 it is `#waitForInput`.

### **(Windows) NotificationManager>>readWinQueue**

This is a mini equivalent of the C-based message loop for Windows shown earlier. It does the following:

1. Peek for a message. In Windows, this also lets the application yield control if no messages are pending. If there is a message, remove it from the queue.
2. Do some special case handling for dialogs and accelerators
3. Translate the message
4. Dispatch the message

The dispatched message is trapped by the window function for the virtual machine, which forces a call to `#recursiveWinMessage`.

### **(OS/2) NotificationManager>>waitForInput**

It simply peeks for a message in the application queue. If there are none, and there are some other messages pending, it sleeps for 1 second and tries again. Unlike Windows, the preemptive nature of OS/2 means that two separate threads can be used for the event loop and the window function, avoiding the need for the explicit dispatch that is used in `#readWinQueue`.

These methods are also called whenever it is important to poll the system for messages. In particular, two other methods use this: `#consumeInputUntil:` (which traps all input messages), and `#cleanUpAllMessages` (which is called during shutdown of Smalltalk).

## Step 2: Notify a Message

When a message is detected in the application queue, the window function is triggered asynchronously by the operating system, that is, the main event loop is moving on to stage 3 while the window function is still being called.

The window function is hidden in the virtual machine, and is not accessible.

All Smalltalk/V windows share the same window function, because the polymorphic aspects of the windows are handled in Smalltalk directly.

The window function does the following:

1. Stores the message in a global structure used to interface between the window function and Smalltalk. In Windows this is a `WinMessage` (a subclass of `WinStructure`) called `WinMsgST`. In OS/2 this is a mini-queue called `InputEvents`.
2. Directly calls a Smalltalk method called either `#recursiveWinMessage` or `#recursivePMessage`.

**NotificationManager>>recursiveWinMessage**  
**NotificationManager>>recursivePMessage**

This method can be thought of as the ‘accessible’ part of the window function. It

1. Restores the stack and virtual machine state.
2. Fetches the message from the global structure.
3. Finds the appropriate window and passes the message to it.
4. Preserves the stack and machine state, and returns to the operating system with a result.

Steps 2 and 3 are combined in the method called `#notifyRecursive`.

Step 3 must return an answer for the operating system. If the result returned is `nil`, the virtual machine will execute the default window function, otherwise the result should be an integer, with the appropriate value depending on the type of message.

**NotificationManager>>notifyRecursive**

This method

1. Fetches the message from the global message structure.
2. Calls `#notify:` with the message as a parameter.

**NotificationManager>>notify:anInputEvent**

This method is the core method of the event processing system. It is passed an input event when it is called by `#notifyRecursive`. The event object is a **InputEvent** in V/OS2, and a **WinMessage** in V/Win.

1. The method first looks at the window handle stored in the input event and tries to find a matching window with that handle. The **Notifier** maintains a list of all active windows for exactly this purpose. The list is stored in the instance variable **windows**.

The code is something like:

```
findWindow: aWindowHandle
    "Private - Answer the Window whose handle
    is aWindowHandle."
    ^windows at: aWindowHandle ifAbsent: [^nil].
```

2. It then looks at the message number stored in the `InputEvent` and tries to find a matching selector.

The most common associations between numeric message numbers and Smalltalk selectors are stored in an array called **WinEvents** (or **PMEvents**). Some less frequently used messages are stored in a dictionary called **WinEventsExtra** (or **PMEventsExtra**).

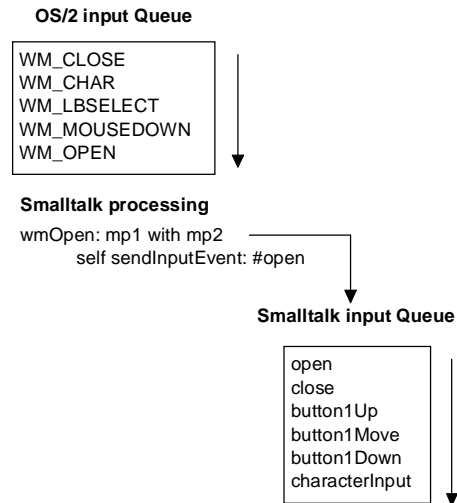
```
selectorFor: aMsgNumber
    "Private - Answer the selector which corresponds
    to the Win message aMsgNumber."
    | selector |
    aMsgNumber <= WinEvents size
        ifTrue: [ "Try WinEvents array"
            selector := WinEvents at: aMsgNumber ]
        ifFalse: [ "Try WinEventsExtra dictionary"
            selector := WinEventsExtra at: aMsgNumber
            ifAbsent:[nil ]].
    selector isNil ifTrue: [^#unknownWinEvent:with:].
    "wasn't found!"
    ^selector
```

For example, the message number for `WM_CHAR` is 122 in OS/2, but 258 in Windows. The `WinEvents` and `PMEvents` arrays both convert the message into the selector **#wmChar:with:**.

3. If the Notifier finds the window and the message selector, the window is asked to perform the message with the two parameters.

```
foundWindow
    perform: selector
    with: (event wParam)
    with: (event lParam).
```

It is good practice for the window function to return as soon as possible so that other applications get a chance to receive message. To do this, the window will typically convert the direct message into an indirect message by adding it to a global message queue which is processed later. This queue should really be called `WindowMessageQueue`, but unfortunately it's confusingly called **CurrentEvents**.



### Converting OS/2 input events into queued Smalltalk events.

The code looks something like this:

```

Window>>wmClose: wordInteger with: longInteger
    "Private - Process the close window message."
    self sendInputEvent: #close. "defer processing"
    ^1                          "return now!"

Window>>sendInputEvent: selector
    "Private - Add a Message to CurrentEvents."
    CurrentEvents add: (Message new
        receiver: self
        selector: selector
        arguments: #() ).
  
```

### Step 3: Process Queued Window Messages

Finally, any queued messages are handled. The way in which these queued messages are handled in V/OS2 and V/Win are completely different, in fact, somewhat arbitrary, and I won't go into too much detail.

There are three queues that are processed:

- **CurrentEvents** has already been mentioned. It contains most of the messages that are deferred by a window.
- **CurrentEvents** are generally processed by the **#empty** method, which loops until there are none left.
- **PendingEvents** contains walkback messages, created by a CNTRL-break, halt, or error. Again, another confusing name! Think of it as the WalkbackMessageQueue.

For example:

```
Process class>>queueWalkback:makeUserIF:resumable:
...some code ...
PendingEvents add: (Message new
  selector: #errorIn:label: ;
  arguments:
    (Array with: process with: aString)).
```

PendingEvents are processed by an instance of InputEvent called CurrentEvent. CurrentEvent is sent the #nullEvent message. If there are any pending events, the first one only is performed. (CurrentEvent is a general helper object for Notifier).

In V/Win, PendingEvents are processed before CurrentEvents and DeferredRequests. In V/OS2, they are processed after.

- DeferredRequests contains low priority messages that must be processed only after all other events have processed. For example, the system shutdown message is deferred.

```
NotificationManager>>reinitialize
"Close all the windows and open a new Transcript."
"Deferred because 'DoIt' or 'ShowIt' need to be
returned"
DeferredRequests add: (Message new
  receiver: self
  selector: #reinitDelayed
  arguments: #() ).
```

**DeferredRequests** are processed only when there are no CurrentEvents.

In V/Win, only one **DeferredRequest** is processed before checking for more **CurrentEvents**. In V/OS2, they are all processed at once.

These queues are processed from within the #run method (or helper methods, such as #runPending) not the window function.

This can lead to synchronization problems, which I discuss next.

## Direct vs. Queued Window Messages

When a window is notified of a message, it can do one of three things:

- A. Post the message to CurrentEvents and return immediately.

This is the preferred option if

- processing the message will take a long time, or
- the message must be executed in sequence, after other messages, or



- one or more low level messages are different between OS/2 and Windows and must be converted to a common ‘event’. (This can also be done by directly calling a method — see #losingFocus).

An example is the #wmClose message shown earlier.

```
Window>>wmClose: wordInteger with: longInteger
    "Private - Process the close window message."

    self sendInputEvent: #close.
    ^1 "return 1 to the window function"
```

- B. Process the message immediately. Don’t post it to the queue.

This is the preferred option if

- processing the message will take a very short time
- the message must be executed urgently.

An obvious example of short processing is the default response to most messages, which is just to return nil.

Another example of short processing is the default response to #wmKillFocus, which is just to send the message #losingFocus.

(Note that in the EntryField subclass, #losingFocus causes a time-consuming validation to occur, so it is converted into a queued message.)

Examples of urgent processing are found in the various wmScroll messages.

- C. A combination of A) and B). Do some preprocessing of the message and, depending on the result, post different messages to the queue.

```
wmChar: wordInteger with: longInteger
    "Private - Process the character input message."
    | char |
    char := wordInteger asCharacter.
    (ControlKeys includes: char)
        ifTrue: [ self sendInputEvent:
            #controlKeyInput: with: char ]
        ifFalse: [ self sendInputEvent:
            #characterInput: with: char ].
    ^nil "also let the system handle it"
```

## Synchronization Problems

The CurrentEvents queue can be quite large. Check the size with:

```
CurrentEvents size
```

If you respond to a message directly, you can often beat an earlier message that was queued. You can demonstrate that this occurs with the following test.

1. First create a list that we will use to store characters.

```
GlobalChars := OrderedCollection new
```

2. Next, refine the **#wmChar:with:** method for Button

```
Button>>wmChar: wordInteger with: longInteger
    "Private - Process the character input message."
    | char |
    char := wordInteger asCharacter.
    GlobalChars add: char.
    ^super wmChar: wordInteger with: longInteger
```

With this refinement, all keystrokes directed to a Button will be added directly to the GlobalChars list.

Test this by setting the focus to a button — the instance radio button in a CHB will do. Then press the ‘q’ key a few times, and inspect GlobalChars. You should see the ‘q’ key in the list of characters.

3. Finally, refine the **#characterInput:** method for Button

```
Button>>characterInput: aChar
    "Private - Process the character input message."

    GlobalChars add: aChar asUppercase.

    10000 timesRepeat: []. "add short delay"

    ^super characterInput: aChar
```

With this refinement, all queued keystrokes directed to a Button will be converted to uppercase and added to the GlobalChars list. The delay simulates the time that would be needed to execute a more complex method.

Reinitialize GlobalChars to an empty collection and retest. Press a sequence of different lowercase keys fast, and inspect GlobalChars. You should see each key repeated, once in lowercase, and then in uppercase.

In particular, some keystrokes are probably out of order, that is, some ‘later’ lowercase keys come before an ‘earlier’ uppercase key.

You can increase this effect by typing faster.

*These synchronization problems are particularly evident in VW 1.x. For example, if you perform a long task, you can still switch between windows, select items from lists, etc., but the effects of these actions are delayed. To see this, change the delay loop from 10,000 to 100,000 and repeat the demonstration above1. You will find that the machine responds like treacle until all the characters are processed.*

## Avoiding Synchronization Problems

- **Be consistent with your event handling technique.**  
Don't mix and match queued and direct methods. In general, *all* methods should be queued.
- **Be aware of which existing methods are queued and which are not.**  
If you are writing an entry-field validation routine for example, it is important to know that `#wmKillFocus` is not queued by default, but `#wmChar` is. If the sequence of events is critical, force all events to be queued (the approach taken in replacing `#wmKillFocus` in `EntryField`).

## Methods Relating to Queued Messages

**remove: setOfEventSelectors for: aWindow**

This removes queued events for aWindow which are in the list of event selectors. For example:

```
Notifier remove: #( wmChar:with: ) for: myWindow.
```

You can use this to resolve certain synchronization problems by removing queued events which should no longer be processed.

**removeEventsFor: aWindow**

This removes *all* queued events for aWindow. For example:

```
Notifier removeEventsFor: myWindow.
```

This is called by `#remove:` before a window is removed from the list of open windows.

## The Notifier and Open Windows

The Notifier maintains a list of open windows so that it can find the window corresponding to a particular window handle. The list is a dictionary with `WindowHandles` as the keys, and window instances as the values. The dictionary is stored in the Notifier instance variable `windows`.

Every new window that is created must be added to the Notifier list, otherwise it will not receive messages. This includes all subpanes, control windows, buttons, etc.

The `#buildWindow` method for `ApplicationWindow` does this. It adds the main application window and recursively adds its subpanes.

When a window is closed it must be removed from the Notifier list before the image is saved. Again, this includes various subpanes.

The `#close` method for `ApplicationWindow` does this. It removes the main application window and its subpanes.

## Reinitializing the Notifier

Because the Notifier knows about all the open windows, it can be used to close them all, and ‘restart’ the user interface. This is the method `#reinitialize`.

### `reinitialize`

This method

- closes all open windows
- creates a new Transcript
- restarts the user interface process.

Reinitialize should be used whenever you make a sweeping change that affects all windows, such as adding a new menu option in the File menu.

In some cases, you may need to perform an operation with no open windows, such as adding a new instance variable to `SubPane2`. You can do this by closing all the windows, doing the operation, and then calling `#reinitialize`.

```
Notifier closeAllWindows. "kill all Window instances"
Window subclass: SubPane "modify SubPane definition"
    instanceVariables: '....'
    classVariables: '....'
    poolDictionaries: '....'
Notifier reinitialize. "start up again"
```

**NOTE:** If you need to add instance variables to a class with instances, there are other, better, ways (for example, use a variable dictionary, like `SubPane properties`).

## Methods Relating to the List of Windows

### `windows`

Returns the collection of open windows.

**add:** *aWindow*

Adds aWindow to the list so aWindow can receive incoming events.”

**remove:** *aWindow*

Calls #removeEventsFor: to kill outstanding events, then removes aWindow from the list of open Windows.

**cleanUpWindows(Win)**

This method cleans out ‘bad’ windows which no longer have valid window handles

**aboutToSaveImage**

This method loops through all the *open* windows and gives them a chance to save their data.

**closeAllWindows(Win)**

This closes all the open windows *including* the Transcript, in preparation for exiting Smalltalk. It is also used when reinitializing the Notifier.

**findWindow:** *aWindowHandle*

Return the Window whose handle is aWindowHandle. This is used in the dispatching process mentioned earlier, but it can also be useful for your own methods.

## The Notifier and the User Interface Process

The **Notifier** only interacts with one process, called the **UserInterfaceProcess**.

The **Processor** maintains a list of all other processes and schedules them.

The **UserIF** process is *not* listed in the Processor, and is *not* scheduled. In this sense, it is kept separate from the virtual machine (in V/OS2, as noted earlier, the **UserIF** is actually a separate thread)..

In general, all windows share the same UserIF process, but occasionally a new UserIF process is needed, for example, in modal dialogs, or the debugger.

Every time a new process is created that wants to be the exclusive user interface process, **#run** must be called. The **#run** method drops the current process chain and restarts the user interface process with ‘no history’.

```
process := Process new.    "create a new process"
process makeUserIF.        "make it the user interface"
Notifier run               "drop process history, and
                           restart the event loop"
```

## Modal Windows

A modal window does not allow other windows to receive input. In Windows and OS/2 there are special functions that allow you to do this easily.

**In Smalltalk these modal functions are *not* used**, because Smalltalk would lose control of the message queue. Instead, modality is implemented the ‘hard’ way, by freezing the current process.

When a modal window is created, the steps taken are:

1. The parent window is disabled
2. a new UserIF process is created
3. the existing UserIF process is put on hold
4. the Notifier is restarted by calling **#run**.

The code for processing modal dialog windows looks like:

```
DialogTopPane>>processInput
  "Private - Make the receiver modal to its
  owner window. The parent should have been
  disabled before calling this. This method
  doesn't return until close has been sent to
  the receiver."
  sem := Semaphore new.

  [CurrentProcess makeUserIF.
  Notifier run] fork
  "create a new UserIF process and restart Notifier"

  sem wait. "put the current process on hold"
  CurrentProcess makeUserIF
  "restart the current process"
```

When the modal window closes

- the semaphore is released and set to nil.
- the active process is shut down, allowing the original UserIF process to continue on its merry way.

```
DialogTopPane>>close
  "Private - Close the dialog box."
  ...some code ...

  sem notNil ifTrue: [
    sem signal. "release the semaphore"
    sem := nil.
```

```
Processor suspendActive  
"kill the dialog process"  
].
```

```
... some more code ...
```

Although complicated, this way of handling modal windows is very flexible. You can have modal windows which are normal application windows, not just special ‘dialog’ boxes. If you want to get weird, you can even mess with the processes and semaphores and toggle modality on and off in mid-stream!

## Trapping User Input Outside a Window

On certain occasions, you will want to trap all user input and direct it to a particular window. This is especially common when using the mouse to draw, resize, or drag objects, and you want to track the mouse until the button is lifted. The Notifier implements the `#consumeInputUntil:` method for this purpose.

**consumeInputUntil:** *aBlock*

This method pulls a message off the `CurrentEvents` queue and passes it to `aBlock` until `aBlock` evaluates true, when the method returns.

`aBlock` takes one argument, which is a `Message`.

To see an example, look at `Screen>>rectangleFromUser`.

Try

```
Display rectangleFromUser
```

**IMPORTANT NOTE:** This only traps *queued* messages from `CurrentEvents`. Direct messages such as scrolling are *not* detected. If you try to trap these kinds of events, you will hang the system!

## Global Objects Relating to the Notifier

The names of the various ‘events’ and ‘messages’ are confusing. Here is a useful table.

Smalltalk/V name	Should really be called!	Description
CurrentEvents	WindowMessageQueue	A queue to store window messages so that a window can return immediately from a notification.
CurrentEvent	InputEventInstance	An instance of InputEvent that has miscellaneous methods to help the Notifier.
PendingEvents	WalkbackMessageQueue	A queue to store walkback events generated by halt, control-break, errors, etc.
DeferredRequests	LowPriorityMessageQueue	A queue to store messages that must be run after all other messages are processed.

Other globals are:

### **WinEvents/PMEvents**

An array of the events with a low numeric value. Each position stores the corresponding selector, or **nil**.

### **WinEventsExtra/PMEventsExtra**

A dictionary of less frequent events. Each key is a message number, and the value is the corresponding selector.

### **OldScreenMode (Win)**

### **PreviousScreenMode (Win)**

### **MachineState(OS/2)**

These are used to save the virtual machine state for startup, and to preserve it whenever control returns to the operating system.



**WinMsgNT (Win)****QMsg (OS/2)**

A WinMessage/PMLong used for peeking at the message queue. The value is never used.

**WinMsgST(Win)**

A WinMessage used for fetching a message from the message queue. The value is passed as a parameter to #notify.

**InputEvents (OS/2)****InputEventsNum (OS/2)**

InputEvents is a small buffer for input events. InputEventsNum is the index of the next event in the buffer. The input event at that index is passed as a parameter to #notify.

**PoppedModelessWindows (Win)**

In Windows, keyboard input for modeless dialog boxes must be handled specially in #readWinQueue. This variable contains a list of all dialog windows that are in use.

It is added to when a dialog is created (for example, DialogBox>>fromModule:id:) and removed from when a dialog is closed.

# Index

## —A—

- abortChange, 129, 219
- abortClose, 287
- About, 78, 86
- aboutToChange, 157, 162, 168, 194, 199, 210, 251
- aboutToChangeLabel:to:, 203, 271
- aboutToChangeTo:, 174, 180, 227, 256
- aboutToClose, 292
- aboutToEditLabel:, 203, 271
- aboutToSave, 259
- Accelerator Keys, 47
- Access, 320, 323
- Accessing a menu programatically, 337
- Accessing a widget programatically, 325
- activated, 292
- Active Menu, 302
- activeCategory, 319, 321
- activeCategory:, 319, 321
- activeTextPane, 287
- Adapting domain models to widgets, 347
- Add, 98, 102
- Add Custom Widget, 355
- Add Menu, 122
  - Add Custom Widget, 123
  - Button, 122
  - Composite, 122
  - Custom Widgets, 123
  - Group, 122
  - List, 122
  - Misc, 122
  - Remove Custom Widget, 123
  - Text, 122
  - Valuator, 122
  - Windows 95, 122
- Add text to a widget, 8
- add:, 279
- add:rbSelector:, 279
- add:selector:, 279
- add:selector:enable:, 279
- add:selector:enable:owner:, 279
- add:selector:enable:owner:spaces:, 280
- add:selector:enable:owner:spaces:rbSelector:, 280
- add:selector:owner:, 280
- add:selector:owner:spaces:, 280
- add:selector:owner:spaces:rbSelector:, 280
- add:selector:rbSelector:, 280
- add:selector:spaces:, 280
- add:selector:spaces:rbSelector:, 280
- add:spaces:, 280
- add:spaces:rbSelector:, 281
- addBorderStyle, 129
- addCategory:, 319, 321
- addClipchildrenStyle, 129
- addClipsiblingsStyle, 129
- addDialogBorderStyle, 287
- addField:, 245
- addGroupStyle, 129
- addHorizontalScrollbarStyle, 129
- Add-In Manager, 76, 117
- Add-In Modules, 366–70
- Adding a timer to a window, 335
- Adding a widget to a window
  - dynamically, 334
- Adding menu items dynamically, 340
- addMaximizeButtonStyle, 287

- addMaximizeStyle, 287
- addMinimizeButtonStyle, 287
- addMinimizeStyle, 287
- addModalBorderStyle, 287
- addObject:, 143
- addPage:, 247
- addPoolNamed:, 318
- addSizingBorderStyle, 287
- addSubpane:, 288
- addSubpaneDynamically:, 288, 334
- addSystemMenuStyle, 288
- addTabStopStyle, 129
- addTick:, 264
- addTitleBarStyle, 288
- addTransparentStyle, 129
- addVerticalScrollbarStyle, 129
- Align Bottom, 106
- Align Left, 106
- Align Menu, 16, 106
- Align Right, 106
- Align Top, 106
- Aligning Widgets, 21
- alignLeft, 200
- alignTop, 200
- allowDuplicateKeys, 319
- allowDuplicateKeys:, 319
- Alpha, 175
- AlphaNoSpace, 175
- AlphaNumeric, 175
- AlphaNumericNoSpace, 175
- altKeyInput:from:, 129
- animate:, 143
- AnimationPane, 125, 143–45
- Any, 175
- append:, 219
- appendPage:, 247
- ApplicationCoordinator, 9, 61, 75, 80, 83, 86, 316, 317
- Asking a yes/no question, 341
- asParameter, 130
- associationAt:, 320, 323
- associationAt:ifAbsent:, 323
- associationsDo:, 320
- at:, 320, 323
- at:ifAbsent:, 320, 323
- at:in:, 321, 323
- at:in:ifAbsent:, 321, 323
- at:in:ifAbsentPut:, 323
- at:in:put:, 321, 323
- at:in:put:ifAbsent:, 321
- Attach a Callback, 9
- Attribute Editor
  - Creating, 362
- Attribute Panel, 6, 14
- attributeEditor, 365
- Attributes, 101
- Attributes Menu, 93
  - Attributes, 101
  - Call Outs, 103
  - Color, 94
  - Events, 96
  - Font, 93
  - Framing, 94
  - Menus, 95
  - NLS, 104
    - NLS Autosize, 104
    - NLS Manager, 104
    - Set Pools, 104
  - PARTS Interface, 105
  - Styles, 100
  - Tabbing/Groups, 101
- Auto Recognize OK and Cancel, 114
- Auto Save, 112, 114
- Auto Size, 113, 114
- Auto Size Selection, 17, 108
- Auto Update Outboards, 114
- autoArrange:, 200
- autoCheckBox, 148, 261
- autoHScroll, 170, 178, 219, 252
- autoRadioButton, 214
- Autosize, 356
- autoSizeFrom, 362
- autoTabStyle, 170
- autoThreeState, 261
- autoVScroll, 170, 178, 219, 252

## —B—

- backColor, 130, 288, 361
- backColor:, 130, 288, 327, 333
- backTabbed, 140
- bitmap, 184
- Bitmap Editor, 305
- Bitmap Manager, 76, 118, 304–11, 366
- Bitmap Menu, 309
  - Copy, 309
  - File In Bitmap File, 311
  - File Out 4 Bit Bitmap File, 311
  - File Out 8 Bit Bitmap File, 311
  - File Out Bitmap File, 311
  - From Screen, 310
  - From Screen Into Button, 310
  - From Screen Into Button Down, 310
  - From Screen Into Button Up, 310
  - Paste, 309
  - Paste Ino Button, 310
  - Paste Ino Button Down, 310
  - Paste Ino Button Up, 310
- bitmap:, 184
- bitmaps, 190, 205
- bitmaps:, 190, 205
- blackFrame, 239
- blackRectangle, 239
- Boolean, 175
- Bottom, 98
- boundingBox, 130
- Bring Forward, 89
- Bring To Front, 89
- Bringing a window to the front of other windows, 332
- bringToFront, 288, 332
- bringToTop, 130
- Button, 8, 17, 125, 146–47
- Button Edit, 366
- Button Editor, 305
- button1Down, 144, 185
- button1DownShift, 144, 185
- button1Moved, 144, 185
- button1UpShift, 144, 185
- button2DoubleClicked, 144, 185

- button2Down, 144, 185
- button2Moved, 144, 185
- ButtonListBox, 125, 153–58
- buttons:, 187, 247

## —C—

- Cache Property Managers, 114
- Call Out Editor, 35, 103
- Call Outs, 35, 103
- Callbacks, 7
  - attaching, 9
- cancelPushButton, 146
- captureMouseInput, 130
- case, 170
- case:, 170
- Categories, 319, 321, 322
- category, 371
- Category Menu, 300, 306, 313, 317
  - Add Suggested, 301
  - Delete, 302
  - File Out, 306, 313, 317
  - File Out Separately, 317
  - New, 300
  - Rename, 301
  - Show All, 302
  - Show Selected, 302
- cellHeight, 281
- cellSize, 281
- cellSize:, 281
- cellWidth, 281
- Center Horizontally, 106
- Center vertically, 106
- centered, 171, 219, 234, 240, 252
- centeredOnMouse, 55
- centeredOnScreen, 55
- Change Grid Size, 22
- changeColor, 219
- changed, 140
- changed:, 157, 162, 169, 174, 180, 194, 199, 203, 210, 213, 227, 230, 237, 251, 256, 259, 266, 271, 275, 288
- changed:with:, 288
- changed:with:with:, 288

- changedIndex:, 157, 194, 199, 210, 217, 237
- changedLabel:to:, 203, 271
- changedPageNumber:, 251
- changeFont, 219
- changeItem:label:, 338
- changeParagraph, 220
- changesSize, 365
- changesText, 365
- changeTabs, 220
- Changing a menu item's label, 338
- Changing a widget's color, 327
- Changing a widget's font, 327
- Changing a widget's label or contents, 328
- Changing a widget's size and position, 326
- Changing a window's bgcolor, 333
- Changing a window's label, 334
- Changing a window's size and position, 332
- changing:, 237, 266, 275
- changingIndex:, 237
- character, 171
- character:, 171
- charactersBeforeLine:, 220
- characterTyped, 196
- characterTyped:, 157, 162, 169, 194, 199, 203, 210, 272
- check, 148, 261
- CheckBox, 70, 85, 125, 148–49, 148, 261
- CheckBoxGroup, 125, 150–52
- checked, 149, 263
- checked:, 157
- checkedIndex:, 157
- checkIndex:, 153
- Checking and unchecking menu items, 339
- checkItem:, 153, 339
- childAtId:, 288
- children, 130, 288
- childrenSize, 130, 289
- Class Hierarchy Browser, 9, 10
- Clear, 89, 99, 143, 159, 166, 171, 178, 220, 252
- Clear All, 102
- clearMessage, 220
- clearModified, 220
- clearMouseCapture, 130
- clearSelection, 153, 159, 166, 171, 178, 190, 196, 205, 220, 252
- clearTextSelection, 159, 166
- click, 146, 148, 164, 214, 261
- clicked, 147, 165
- clicked:, 144, 149, 151, 157, 162, 169, 185, 188, 194, 199, 203, 210, 215, 217, 262, 282
- clickedIndex:, 188, 203
- closed, 292
- Closing unwanted dialogs, 340
- Code Generation, 371–72
- collapsed:, 272
- collapseItem:, 269
- Color, 94, 357
- Color Editor, 28, 29
  - system, 29
- colorDefaults, 357
- colorElements, 357
- Colors, 28
- columnClicked:, 203
- columnWidths, 187
- ComboBox, 34, 125, 159–63
- Comment Methods, 114
- Company Name, 115
- CompositePane, 61, 75, 80, 83, 86, 81–86, 352
  - Creating, 295
  - Events, 296
  - Nesting, 295
  - Tab Order, 296
- confirm:, 341
- consumeInputUntil:, 342
- contents, 130, 146, 148, 150, 153, 159, 164, 166, 171, 178, 182, 187, 190, 196, 200, 205, 214, 216, 220, 232, 234, 240, 242, 245, 247, 252, 257, 261, 269, 276, 277, 361

contents:, 130, 143, 146, 148, 150, 153,  
 160, 164, 166, 171, 178, 182, 187,  
 190, 196, 201, 205, 214, 216, 220,  
 232, 234, 240, 242, 245, 247, 252,  
 257, 261, 269, 276, 277  
 ControlPane, 129–42  
 controlTabbed, 227  
 Cookbook, 325–54  
 Cooper & Peters, 2, 38  
 Copy, 88, 99  
 Copyright, 115  
 Copyright After Body, 115  
 copySelection, 159, 166, 171, 178,  
 220, 252  
 copySpecificsTo:, 363  
 cr, 220, 257  
 Create, 92  
 Create All, 103  
 Create Bitmap Dialog, 307  
 Create Button Dialog, 307  
 createMenus:, 52, 60  
 createViews, 52, 60, 76, 80, 103  
 Currency, 176  
 CurrencyNoDecimal, 176  
 currentIndex, 281  
 currentPage, 248  
 cursorWindowPosition, 130, 289  
 Custom Widgets, 123  
 Customizing WindowBuilder Pro, 328–  
 71  
 Cut, 88, 99  
 cutSelection, 159, 166, 171, 179, 220,  
 253

## —D—

Date, 176  
 deactivated, 292  
 deanimate:, 143  
 decrementPosition, 212, 264, 273  
 decrementPositionBy:, 212, 229, 264,  
 273  
 Default Font, 357  
 defaultBackColor, 130, 357

defaultCursor, 130  
 defaultFont, 131, 358  
 defaultForeColor, 131, 357, 358  
 defaultMessageArguments, 371  
 defaultMessageSelector, 371  
 defaultPushButton, 146, 358  
 defaultStyle, 358  
 defaultWizardEvent, 359  
 deleteAll, 153, 160, 166, 167, 187,  
 190, 196, 201, 205, 220, 234, 253,  
 269, 339  
 deleteAllColumns, 201  
 deleteAllPages, 248  
 deleteIndex:, 153, 160, 167, 190, 196,  
 205, 234  
 deleteItem:, 153, 160, 167, 190, 196,  
 205, 234, 269  
 deletePage:, 248  
 deletePageNumber:, 248  
 deleteText, 220, 253  
 demandLoad, 248  
 demandLoad:, 248  
 deselect, 154, 191, 197, 206  
 deselectAll, 206  
 deselectIndex:, 206  
 deselectItem:, 206  
 Dialogs, 340  
 disable, 131, 289, 329, 333  
 disabled, 131, 289  
 disableElements, 281  
 disableItem:, 281, 338  
 disableNoScroll, 154, 191, 206  
 disableRedraw, 131, 289  
 disableUpdate, 131, 289  
 disableWordWrap, 221, 253  
 display, 140, 144, 185  
 Display Grid, 23  
 Displaying a message, 341  
 displayWith:clipRect:, 360, 361  
 Distribute, 19  
 Distribute Horizontally, 106  
 Distribute Vertically, 106  
 Distributing Widgets, 19  
 dividerDoubleClicked:, 188

- dividerDoubleClickedIndex:, 188
- Do It, 99
- doubleClicked, 144, 185, 243
- doubleClicked:, 157, 163, 194, 199, 203, 210, 272, 282
- doubleClickedIndex:, 203
- Down, 98, 237
- Drag Drop Handlers, 100
- Drag Outlines, 115
- dragSessionClass:, 131
- dragSource, 131
- dragSource:, 131
- dragSourceCut:, 140
- dragSourceNeedsObject:, 140
- dragTarget, 131
- dragTarget:, 131
- dragTargetDrawEmphasis:, 141
- dragTargetDrop:, 141
- dragTargetEmphasisDefault, 131
- dragTargetEmphasisItem, 131
- dragTargetEmphasisPane, 131
- dragTargetEmphasisSeparator, 132
- dragTargetEnter:, 141
- dragTargetEraseEmphasis:, 141
- dragTargetForFormats:operations:, 132
- dragTargetLeave:, 141
- dragTargetMultipleItem, 132
- dragTargetMultipleItem:, 132
- dragTargetNeedsOperations:, 141
- dragTargetOver:, 141
- Draw Grid, 23, 109, 115
- drawBitmap:for:, 191, 206
- drawBox, 132
- drawFocus:, 194, 210, 292
- drawIndex, 132
- drawingRectangle, 132
- drawItem, 293
- drawItem:, 194, 210, 246, 251
- DrawnButton, 126, 164–65
- drawSelection:, 194, 210, 293
- dropDown, 160
- DropDownList, 126, 160, 166–69
- Duplicate, 89
- Dynamic CompositePanels, 352

## —E—

- edit, 38, 321
- Edit an Existing Window, 13
- Edit Class Dialog, 13
- Edit Menu, 88
  - Bring Forward, 89
  - Bring To Front, 89
  - Clear, 89
  - Copy, 88
  - Cut, 88
  - Duplicate, 89
  - Event Summary, 90
  - Morph, 89
  - Paste, 88
  - Paste Window Bitmap to Clipboard, 91
  - Redo, 88
  - Select, 89
    - All, 89
    - All In Same Class, 89
    - All In Same Hierarchy, 89
  - Send Backward, 89
  - Send To Back, 89
  - Test Window, 91
  - Undo, 88
  - Undo/Redo List, 88
- Edit Models, 82
- Edit Window, 75
- editItemLabel:, 269
- editLabels:, 201, 269
- enable, 132, 289, 329, 333
- enableElements, 281
- enableItem:, 281, 338
- enableRangeSelection, 264
- enableRedraw, 132, 289
- enableUpdate, 132, 289
- enableWordWrap, 221, 253
- Enabling and disabling menu items, 338
- Enabling and disabling widgets, 329
- Enabling and disabling windows, 333
- end, 230, 267

- EnhancedEntryField, 70, 85, 126, 170–77
  - ensureVisible:, 270
  - entered:, 174, 181, 256
  - entireClientArea, 248
  - EntryField, 8, 126, 160, 178–81, 355
  - EntryFieldGroup, 126, 182–83
  - ENVY/Developer, 3, 14, 63, 81, 86
  - erase, 184
  - Event Manager, 9, 36–39, 37, 69, 96
    - edIt integration, 39
    - Show All, 38
  - Event Mappings, 77
  - Event Summary, 90
  - Event Translation, 77
  - EventManager, 61, 83, 84
  - Events, 96
  - Examples, 79
  - Exit, 86, 102
  - expanded:, 272
  - expandItem:, 270
  - expressionEvaluator, 257
  - expressionEvaluator:, 257
  - extendedSelect, 206
  - extent, 132
  - Extra Event List Editor, 62
- F—**
- field, 171
  - field:, 171
  - fieldClass, 182
  - fieldNameed:, 245
  - File Menu, 80
    - About, 86
    - Browse Class, 90
    - Browse Widget Class, 90
    - Composite Panes, 81
      - Ungroup, 81
    - Exit, 86
    - Installed Products, 87
    - Layout Wizard, 84
    - Model Objects, 81
      - Edit Models, 82
      - Model Wizard, 83
        - Select Primary Model, 82
    - New Dialog, 80
    - New Window, 80
    - Open, 80
    - Save, 85
    - Save As, 85
    - Save As Default, 86
    - WindowPolicies, 83
  - fileException:, 227
  - fileInFrom:, 257
  - fileOutOn:, 257
  - Finding the window under the pointer, 337
  - first, 234
  - firstPage, 248
  - fixedSize, 164, 242, 277
  - fixedWidth:, 248
  - Font, 93, 132, 361
  - Font Editor, 28, 93
    - standard, 27, 313
  - Font Manager, 76, 118, 312–14
  - font:, 132, 327
  - Fonts, 27
  - forceEndOntoDisplay, 221, 257
  - forceSelectionOntoDisplay, 221, 258
  - Forcing a widget to redraw, 330
  - foreColor, 133, 289, 361
  - foreColor:, 133, 289, 327
  - Fork with interrupts, 343
  - formatRect, 221, 253
  - formatRect:, 221, 253
  - formattedContents, 221
  - formattedContents:, 221
  - frameRectangle, 133, 289
  - frameRelativeRectangle, 133, 289
  - frameWindow, 133
  - Framing, 94
  - Framing Editor, 31, 94
  - Framing Styles, 34
  - framingBlock, 326, 332
  - framingBlock:, 133, 326, 332
  - framingRatio:, 133
  - freeClientArea, 248



**—G—**

- General, 321, 324
- Generate Byte Array Code, 115
- Generate Copyright, 115
- Generate Portable Font Code, 115
- Generate SysFont References, 115
- generateBody, 371
- generateComment, 371
- generateTemporaries, 371
- getAlignmentFor:, 187
- getBorderSizes, 245
- getChildrenExpandedFor:, 270
- getChildrenFor:, 270
- getColumnsFor:, 201
- getFocus, 171
- getFocus:, 171
- getHasChildrenFor:, 270
- getImageFor:, 154, 187, 201, 270
- getItemHeight, 248
- getItemRect, 248
- getItemRect:, 248
- getItemWidth, 248
- getMnemonicHandler:, 133
- getSelectedImageFor:, 270
- getSelection, 171
- getSmallImageFor:, 201
- getStateFor:, 154
- getStringFor:, 154, 187, 201, 270
- getStyle, 133
- getTabStop, 221, 253
- getText, 221
- gettingFocus, 141
- getTopIndex, 154, 191, 206
- getValue, 133
- getWidthFor:, 187
- go, 143
- Graphic Object Naming, 355
- GraphicObject, 362
- GraphicObject Framework, 355
- GraphPane, 126, 184–85
- grayFrame, 239
- grayRectangle, 239
- Grid, 22

- Changing Size, 22
- Grid Size, 115
- GroupBox, 126, 186–89
- Groups Menu, 102
  - Add, 102
  - Create All, 103
  - Include Outer, 103
  - Remove, 102
- growTextLimit, 221

**—H—**

- Handle, 7
- Handle Size, 115
- Handler Menu, 98
  - Add, 98
  - Bottom, 98
  - Down, 98
  - Remove, 98
  - Show All, 99
  - Top, 98
  - Up, 98
- hasButtons:, 270
- hasDescendant:, 133
- hasFocus, 133
- hasLines:, 270
- hasLinesAtRoot:, 270
- hasSmalltalkMenuBar, 289
- hasStyle:, 133
- hasTransparentStyle, 134
- Header, 126, 187–89
- headings, 188, 201
- headings:, 188, 201
- height, 134, 245, 290
- height:, 242
- Hello, World, 8
- help, 141, 293
- hideWindow, 134, 290, 328, 331
- Hiding and showing widgets, 328
- Hiding and showing windows, 331
- hitRect, 362
- home, 230, 267
- horizontal, 229, 264, 273



- icon, 290
- icon:, 290, 336
- iconView, 201
- id, 134
- impatient users, 342
- Include Outser, 103
- includeSizeGrip:, 245
- includesKey:, 319, 322
- incrementPosition, 212, 264, 273
- incrementPositionBy:, 212, 229, 264, 273
- indent, 270
- indent:, 270
- indeterminate, 262
- indeterminate:, 157
- indeterminateIndex:, 154, 157
- indeterminateItem:, 154
- indexOf:, 150, 154, 160, 167, 191, 197, 206, 216, 234
- Indirect Bitmap References, 116
- Indirect Font References, 116
- Indirect NLS References, 116
- initialize, 83
- initializeCodeGeneration, 371
- initializeForDataType:, 359
- initialSize:, 290
- initWindow, 52, 364
- Input Focus, 357
- inputOccurred, 251
- insert:, 221, 253, 258
- insertAfterSelection:, 221, 258
- insertAndSelect:, 221, 258
- insertItem:, 154, 160, 167, 191, 197, 206, 234
- insertItem:at:, 154, 160, 167, 191, 197, 206, 235
- insertItemDynamically:selector:  
atIndex:, 340
- insertPage:at:, 249
- insertPageAtSelection:, 249
- insertSelectedText:at:, 172, 179, 222, 253
- inset, 242
- inset:, 242
- Inspect It, 99
- Installation, 5
- Installed Products, 78, 87
- Integer, 176
- integralHeight, 154, 191, 206
- invalidate, 134
- invalidateRect:, 134
- invalidateRect:erase:, 134
- invisible, 164
- isActive, 134
- isActive:, 143
- isAtTop, 245
- isDragButton:, 134
- isDragSource, 134
- isDragTarget, 134
- isFirstPage, 249
- isGlobalDragSource, 134
- isGroupable, 361
- isHandleOk, 134
- isHidden, 135
- isHorizontal, 265, 273
- isLastPage, 249
- isLeftJustified, 242
- isLocalDragSource, 135
- isMaster, 235
- isModified, 222
- isOffScreen, 135
- isReadOnly, 222
- isResizable, 242
- isRightJustified, 242
- isTextPane, 222
- isValid, 135
- isVertical, 265, 273
- isVisible, 135
- isWordWrapEnabled, 222
- itemHeight, 154, 191, 206
- itemHeight:, 154, 191, 206
- itemIndexFromPoint:, 155, 191, 197, 207
- items, 201
- itemsAttribute, 270
- itemStates:, 155

**—J—**

justified, 243

**—K—**

Key Menu, 302, 306  
  Browse References, 302  
  Delete, 302  
  Duplicate, 307  
  File In Bitmap File, 308  
  File Out BDT File, 308  
  File Out RC and 4 Bit Supporting Files, 308  
  File Out RC and 8 Bit Supporting Files, 308  
  File Out RC and Supporting Files, 308  
  From Clipboard, 307  
  From Screen, 308  
  Move, 302  
  New, 302, 306  
  New Button, 307  
  Rename, 302  
  Resize, 308  
  Resize Button, 308  
keyAtValue:, 320, 322  
keyAtValue:ifAbsent:, 320, 322  
keyAtValue:in:, 322  
keyAtValue:in:ifAbsent:, 322  
keys, 320, 322

**—L—**

label, 135, 290  
label:, 146, 148, 150, 164, 182, 186, 216, 262, 290  
labelWithoutPrefix:, 290, 334  
last, 235  
lastPage, 249  
Layout Pane, 6  
Layout Wizard, 61, 69, 84, 359  
  example, 71  
  generating default layouts, 70

layoutFrame, 135  
layoutFrame:, 135  
LayoutFrames, 31  
left, 172  
leftJustified, 222, 235, 240, 243, 253  
leftJustifiedFixed, 243  
leftJustifiedNoWordWrap, 240  
leftJustifiedWordWrap, 240  
Line Before Comment, 116  
lineAt:, 155, 191, 197, 207, 235  
lineIncrement, 229, 265, 273  
lineIncrement:, 212, 229, 265  
list, 150, 155, 160, 167, 191, 197, 207, 216, 235  
list:, 151, 155, 161, 167, 191, 207, 217, 235  
ListBox, 34, 126, 190–95  
listFont:, 135  
ListPane, 126, 196–99  
ListView, 126, 200–204, 201  
listVisible, 169  
load:, 318  
Log Scratch Window Code, 116  
losingFocus, 141  
lowerCase, 172, 179, 222, 253

**—M—**

mainWindow, 135  
Make Callbacks Private, 116  
makeActive:, 144  
manager, 324  
Marquee selection method, 17  
master, 235  
master:, 235  
matchPatternBack:from:, 222  
Max Undo Levels, 116  
Max Window Size, 116  
maximized, 293  
Maximizing a window, 336  
maximum, 235, 265, 273  
maximum:, 212, 229, 235, 265, 274  
maximumSize, 290  
maximumSize:, 290

- maxSize, 356
  - maxTextHeight:, 222, 253
  - maxTextWidth:, 222, 253
  - mayUndo, 222, 254
  - MDI, 331
  - MDIFrame, 331
  - measureItem, 293
  - measureItem:, 195, 210
  - Menu Bar, 6
  - Menu Editor, 50
  - Menu Reference, 75–86
  - Menubar
    - Creating, 45
  - Menubar Editor, 45, 95
  - menuBarBuilt, 293
  - Menus, 45, 95, 337
    - Adding Menu Items, 46
    - Adding Popup Menus, 49
    - Adding Separator lines, 47
    - Adding Submenus, 46
    - Adding Titles, 46
    - Creating WindowPolicies, 48
    - Editing, 47
    - Keyboard Accelerators, 47
    - Mnemonics, 47
    - Selectors, 48
    - Testing, 48
  - menuTitled:, 290, 337
  - message:, 341
  - messageNote:, 341
  - Method Menu, 99
    - Clear, 99
    - Copy, 99
    - Cut, 99
    - Do It, 99
    - Inspect It, 99
    - Paste, 99
    - Save, 99
    - Show It, 99
  - minimized, 293
  - Minimizing a window, 335
  - minimum, 235, 265, 274
  - Minimum and Maximum Size, 356
  - minimum:, 212, 229, 235, 265, 274
  - minimum:maximum:, 212, 265, 274
  - minimum:maximum:position:, 212, 265, 274
  - minimumSize, 290
  - minimumSize:, 290
  - minSize, 356
  - Mnemonic Keys, 47
  - Mnemonic redundancy checking, 39
  - mnemonic:typedIn:, 135
  - Model Object Editor, 67, 69
  - Model Objects, 61–71
    - attaching to windows, 67
    - creating, 62
    - defining attributes, 63
    - primary models, 68
  - Model Wizard, 61, 69, 75, 83, 84
  - modified, 135, 222
  - modified:, 172, 179, 254, 258
  - modifiedIsTrue, 227
  - Morph, 89
  - Morphing, 34, 359
  - mouseLocation, 184
  - mouseMoved:, 141, 145, 185
  - Move a Widget, 8
  - Move By Pixel, 16, 106
  - multiColumn, 192, 207
  - multipleLines:, 249
  - multipleSelect, 207
  - MultipleSelectListBox, 126, 205–11
  - mutateSpecificsFrom:, 359
  - mutationException, 359
  - mutationExceptions, 359
  - mutationTypes, 359
- N—**
- name, 135
  - Name a Widget, 8
  - Naming panes, event handlers and
    - other methods, 349
  - Naming Widgets, 54
  - National Language Support, 104, 299, 315
  - needsAlignmentFor:, 188

- needsChildrenFor:, 272
- needsColumnsFor:, 203
- needsContents, 142, 174, 181, 227, 241, 256, 259
- needsHasChildrenFor:, 272
- needsImageFor:, 158, 188, 203, 272
- needsMenu, 142
- needsPopupMenu, 142
- needsSelectedImageFor:, 272
- needsSelection, 142
- needsSmallImageFor:, 203
- needsStateFor:, 158
- needsStringFor:, 158, 188, 203, 272
- needsWidthFor:, 188
- New CompositePane, 75
- New Dialog, 75, 80
- New Event Model, 77, 346
- New Model Dialog, 63
- New Window, 75, 80
- nextLine, 230, 237, 267, 275
- nextPage, 230, 249, 267
- nextPut:, 172, 179, 222, 254, 258
- nextPutAll:, 172, 179, 222, 254, 258
- NLS, 104, 299, 315
- NLS Autosize, 104, 116
- NLS Editor, 104
- NLS Extraction Dialog, 316
- NLS Manager, 76, 104, 119, 315–17
- NLS Replacement Dialog, 317
- noBorders, 184
- noDefaultStyle, 136
- noGroupBox, 182, 217
- noGroupLeader, 136
- noHideSelection, 179, 223, 254
- noIntegralHeight, 192, 207
- noPrefix, 240
- noRedraw, 192, 207
- noScrollBars, 184
- noSmalltalkMenuBar, 291
- noTabStop, 136
- Notifier, 386–404
- notify, 192, 207
- notify:withText:, 341
- numColumns, 151, 217

- numColumns:, 151, 217
- Numeric, 176

## —O—

- objectChanged:, 174
- objectChanging:, 174
- Objectshare Systems, 2
- obsolete code, 345
- oemConvert, 179, 254
- offset, 337
- Old Event Model, 77, 346
- open, 52, 55, 80, 330
- openAsMDIParent, 56, 331
- openCenteredOnMouse, 56
- opened, 293
- Opening a window, 330
- Opening a window as an MDI parent or child, 331
- Opening a window floating above another, 330
- openOn:, 82
- openRelativeTo:offset:, 56
- openWindow, 291
- openWithMDIParent:, 56, 331
- openWithMyParent:, 56
- openWithParent:, 56, 330
- Option Menu, 100, 303, 309, 314
  - Allow Duplicate Keys, 303
  - Drag Drop Handlers, 100
  - Non Portable File Out Format, 314
  - Portable File Out Format, 314
  - Same Bitmap All Categories, 309
  - Show Active Category, 303
  - Show All Buttons, 100
  - Show All Selectors, 100
  - Show Dimensions, 309
  - Show Path, 303
  - Update Source, 309
- Options Menu, 109
  - Auto Save, 112
  - Auto Size, 113
  - Drag Outlines, 111
  - Grid, 109

- Draw Grid, 109
- Set Grid Size, 109
- Use Grid, 109
- Look Policy, 109
  - Default, 109
  - OS/2, 111
  - Windows 3.1, 110
  - Windows 95, 110
- Manager, 117
  - Add-In Manager, 117
  - Bitmap Manager, 118
  - Font Manager, 118
  - NLS Manager, 119
- Properties, 114
- Redraw, 119
- Show Z-Order, 111
- Target Is First, 112
- Templates, 113
- Update Outboards, 112
- Use Fence, 112
- Zoom Layout, 113
- owner, 136, 337
- owner:, 136
- ownerDraw, 249
- ownerDrawFixed, 192, 207
- ownerDrawVariable, 192, 207

## —P—

- pageCount, 249
- pageForNumber:, 249
- pageIncrement, 229, 265
- pageIncrement:, 230, 265
- pageNumber, 249
- pageNumberFor:, 249
- pages, 249
- paneName, 136
- paneName:, 136
- paneNamed:, 55, 325
- parent, 136
- PARTS
  - Workbench Integration, 371–80
- PARTS Interface, 105
- PARTS Support, 116

- Passing Arguments, 52
- Passing Messages, 55
- password, 172, 179, 223, 254
- Paste, 88, 99
- Paste Window Bitmap to Clipboard, 91
- pasteSelection, 161, 167, 172, 179, 223, 254
- PButton, 356, 358
- pen, 136
- PEntryField, 355
- performWhenValid, 136
- Person Editor, 57
- PhoneNumberExtUS, 177
- PhoneNumberUS, 177
- Place a Widget, 7
- placeAtTop, 245
- Placing, 7
  - Multiple Widgets, 17
- pool, 324
- Pool Dictionaries, 360
- Pool Manager Programatic Assess, 318
- Pool Managers, 299–324
- Pool Menu, 300, 306, 313, 316
  - Add Existing, 300
  - Delete, 300
  - Extract Strings, 316
  - File In, 306, 313, 316
  - File Out, 306, 313, 316
  - New, 300
  - Remove, 300
  - Replace Strings, 317
  - Update Dependent Classes, 300
- Pool Programatic Assess, 321
- poolFor:, 318
- poolName, 324
- poolNamed:, 319
- Pools, 318, 319
- popup, 136
- Popup Menus, 49
  - Adding to a Widget, 50
- Popup Widget Menus, 41
- position, 212, 230, 265, 274
- Position a widget, 15
- Position Button, 14

- position:, 213, 230, 265, 274
- positionRelativeTo:offset:, 56
- Positive10Comma10, 176
- PositiveInteger, 176
- postAutomatic, 281
- postEdit:, 365
- Potpourri, 342
- preAutomatic, 281
- preEdit:, 365
- preferredLabelOrientation, 359
- preInitWindow, 52
- previousLine, 231, 237, 267, 275
- previousPage, 231, 249, 267
- previousValue, 161, 172, 179, 223, 254, 258
- previousValue:, 161, 172, 179, 223, 254, 258
- Primary Models, 68
- print, 223
- printOn:title:inset:, 223
- printSelector, 161, 167, 192, 197, 207, 235
- printSelector:, 161, 167, 192, 197, 207, 236
- Programatic Interface, 318–24
- ProgressBar, 126, 212–13
- prompt:default:, 342
- Properties, 76, 114
- Property Editor, 76, 114
- propertyAt:, 136
- propertyAt:ifAbsent:, 136
- propertyAt:ifAbsentPut:, 137
- propertyAt:put:, 137
- PScrollBar, 363
- pushButton, 146, 358

## —Q—

- queryFirstPage, 250
- queryLastPage, 250
- Quick Reference, 40

## —R—

- RadioButton, 126, 213–15, 214
- RadioButtonGroup, 34, 127, 216–18
- raggedRight:, 250
- rbSelectorAt:, 281
- readOnly, 172, 179, 223, 254, 258
- readOnly:, 172, 223, 254
- readSpecificsFrom, 362
- readWrite, 172, 180, 223, 254, 258
- realInvalidateRect:, 137
- receiveAllWindowsMessages, 137
- Record Creator Information, 117
- rect, 360
- rectangle, 137, 250, 291
- rectangle:, 137, 291
- Redo, 88
- redraw, 137, 291, 330
- Reframing Widgets, 31
- Remove, 92, 98, 102
- removeBorderStyle, 137
- removeCategory:, 319, 322
- removeHorizontalScrollbarStyle, 137
- removeItemDynamically:, 339
- removeKey:, 320, 322
- removeKey:ifAbsent:, 320, 322
- removeMaximizeButtonStyle, 291
- removeMinimizeButtonStyle, 291
- removePool:, 319
- removeSizingBorderStyle, 291
- removeSubpane:, 291
- removeSubpaneDynamically:, 291, 334
- removeSystemMenuStyle, 291
- removeVerticalScrollbarStyle, 137
- Removing menu items dynamically, 339
- renameKey:to:, 323
- Replicate Height, 19, 108
- Replicate Width, 19, 108
- Replicating Widget Sizes, 19
- reportView, 201
- Requesting a textual response, 342
- requiredPoolDictionaries, 360
- Reset, 102

resizable, 243  
 Resize a Widget, 8  
 Resize Bitmap/Button Dialog, 309  
 resize:, 326, 332  
 resized, 142, 293  
 resizedIndex:to:, 188  
 resizingIndex:to:, 188  
 restore, 155, 192, 197, 208, 223, 236, 258  
 restored, 293  
 restoreSelected, 155, 192, 197, 208, 236  
 restoreWithRefresh:, 155, 192, 197, 208, 236  
 Restoring a window, 336  
 Retrieve from Scrapbook Dialog, 41  
 retryChange, 173, 180, 223, 255  
 Returning Values, 53  
 Reverse, 102  
 RichEdit, 127, 219–28  
 right, 173  
 rightClicked, 142, 145, 185, 293  
 rightJustified, 223, 236, 240, 243, 255  
 rightJustifiedFixed, 243  
 Round2, 177  
 Round3, 177  
 Rubberband selection method, 18  
 runtime errors, 343  
 Runtime Less Code, 117  
 Runtime-less applications, 344

## —S—

Sample Application, 57–60  
 Save, 10, 85, 99  
 Save As, 85  
 Save As Default, 86  
 saved, 260  
 saveToFile:, 224  
 Scrapbook, 39–41, 120  
     Load, 121  
     Merge, 121  
     New, 121  
     Quick Reference, 121  
     Retrieve, 120  
     Save, 121  
     Store, 120  
 ScrollBar, 127, 229–31  
 scrollBoth, 137  
 scrollBy:scrollRect:clipRect:flags:, 137  
 scrollHorizontally, 138  
 scrollVertically, 138  
 Select All, 17  
 Select All In Same Class, 17  
 Select All In Same Hierarchy, 17  
 Select Primary Model, 82  
 Select Primary Model Dialog, 68  
 selectAfter:, 224, 255, 258  
 selectAll, 173, 180, 224, 255, 258  
 selectAtEnd, 224, 255, 259  
 selectBefore:, 255, 259  
 selectCharFrom:to:, 224  
 selected:, 204  
 selectedIndex, 202  
 selectedItem, 155, 161, 167, 192, 197, 202, 208, 217, 224, 236, 255, 259, 271, 282  
 selectedItems, 151, 208  
 selectedPage, 250  
 selectedText, 224  
 selectFirst, 173  
 selectFrom:to:, 173, 180, 224, 255, 259  
 selectIndex:, 151, 155, 161, 168, 192, 198, 202, 208, 217, 236  
 Selecting Groups of Widgets, 17  
 selecting:, 282  
 selection, 138, 149, 151, 155, 161, 168, 193, 198, 202, 208, 214, 217, 224, 236, 262, 271  
 selection:, 138, 149, 151, 155, 161, 168, 193, 198, 208, 214, 217, 236, 262, 271  
 selections, 151, 208  
 selectItem:, 198, 271, 282  
 selectLast, 173  
 selectLine:, 224  
 selectLineAtChar:, 224  
 selector, 282



- selectorAt:, 282
- selectPage:, 250
- Send Backward, 89
- Send To Back, 89
- Serial Number, 117
- Set Color, 28
- Set Font, 27
- Set Grid Size, 22, 109
- Set Pools, 104
- Set Widget Position, 15, 108
- Set Widget Size, 16, 108
- Set Window Position, 14, 108
- Set Window Size, 15, 108
- setCharColor:, 224
- setCharFont:, 225
- setColumnWidth:, 155, 193, 208
- setContents, 225
- setEvaluate:, 259
- setFocus, 138, 329
- setFocusOnControl, 138
- setFromFile:, 225
- setHorizontalExtent:, 156, 193, 208
- setItemHeight:, 250
- setItemSize:, 250
- setItemWidth:, 250
- setLabel:, 147, 149, 164, 186, 214, 262
- setLabelFont:, 183
- setList:, 156, 161, 168, 193, 198, 208, 236
- setMenu:, 138
- setModified, 225
- setName:, 138
- setPages:, 250
- setParagraphAlignment:, 225
- setParagraphOffset:, 225
- setParagraphOffsetIndent:, 225
- setParagraphRightIndent:, 225
- setParagraphStartIndent:, 225
- setParagraphToBullet:, 225
- setPopupMenu:, 138
- setScroll:min:max:, 230
- setStyle:, 138
- setTabStops:, 156, 193, 209, 225
- setTextLimit:, 173, 180, 225, 255
- Setting a window's icon, 336
- Setting focus to a widget, 329
- setToFemale, 232
- setToMale, 232
- setTopIndex:, 156, 193, 209
- setValue:, 149, 156, 161, 168, 173, 180, 193, 198, 209, 213, 215, 225, 230, 236, 241, 255, 259, 262, 266, 274, 328
- setValueFont:, 183
- setValueIndex:, 156, 162, 168, 193, 198, 209, 236
- setValueIndices:, 209
- sex, 232
- sex:, 232
- sexChanged:, 232
- SexPane, 127, 232–33
- SharedValues, 61
- Shift-Select, 18
- Show All, 99
- Show All Buttons, 100
- Show All Selectors, 100
- Show It, 99
- Show Z-Order, 23, 117
- showCaret:, 138
- showDropdown:, 162, 168
- showHelp:, 245, 282
- showMaximizedWindow, 336
- showMinimizedWindow, 335
- showRestoredWindow, 336
- showSelection, 198
- showSelectionAlways:, 202, 271
- showWindow, 138, 328, 331
- simpleList, 162
- simpleMode:, 246
- simpleModeText:, 246
- Size a widget, 15
- Size Button, 15
- Size By Pixel, 16, 107
- Size Menu, 108
  - Auto Size Selection, 108
  - Replicate Height, 108
  - Replicate Width, 108
  - Set Window position, 108

Set Window Size..., 108  
 smallIconView, 202  
 Smart Set, 102  
 sort, 193, 209  
 sortAscending, 202  
 sortAscending:, 202  
 sortByColumn:, 202  
 sortItems, 202  
 spin:, 236  
 SpinButton, 127, 234–38  
 spinDown:, 237  
 spinnedAround:, 238  
 spinUp:, 237  
 SSN, 177  
 standard, 193, 209  
 Standard Translation, 76  
 Start WindowBuilder Pro, 5  
 startGroup, 139  
 startTimer:period:, 291, 335  
 StaticBox, 127, 239  
 StaticText, 17, 127, 240–41, 365  
 statusBoxAt:, 243, 246  
 StatusPane, 127, 242–44  
 StatusWindow, 127, 245–46  
 staysToBack, 361  
 stopAll, 144  
 stopTimer:, 291, 335  
 Store in Scrapbook Dialog, 40  
 storeSpecificsOn:indentString:, 363  
 stream, 371  
 streamContents, 226  
 stretch, 276  
 stretch:, 184, 276  
 stretchToFit, 165, 277  
 style, 361  
 Style Editor, 30, 100  
 styleMap, 358  
 Styles, 30, 100, 358  
     Changing, 30  
 SubPane, 129–42  
 subPaneWithFocus, 292  
 suggestedSize, 356, 357  
 superWindow, 139  
 supportedInDialogs, 361

Switch To, 92  
 System Bitmaps, 385

## —T—

tab, 226  
 Tab Order, 23  
     automatically establishing groups,  
         26  
     creating groups, 25  
     removing group, 26  
     resizing groups, 26  
     setting, 24  
     smart set option, 25  
 Tab Order Editor, 24, 101, 357  
 Tab Order Menu, 102  
     Clear All, 102  
     Exit, 102  
     Reset, 102  
     Reverse, 102  
     Smart Set, 102  
 tabbed, 142  
 Tabbing/Groups, 101  
 TabControl, 127, 247–51  
 tabGroupMembers, 139  
 tabStop, 139  
 tabStopInterval, 226, 255  
 tabStopInterval:, 226, 255  
 Target Is First, 117  
 targetClass, 372  
 targetObject, 372  
 Team/V, 3, 14, 63, 81, 86  
 Technical Support, 3  
 Template Editor, 113  
 Templates, 76, 113  
 Test Window, 10, 91  
 text, 162  
 Text field, 356  
 text:, 162  
 textChanged:, 163, 174, 181, 227, 256  
 textChanged:field:, 183  
 TextEdit, 127, 252–56  
 textLimit, 226  
 TextPane, 127, 257–60

- threeState, 262
- threeState:, 156
- ThreeStateButton, 127, 261–63
- threeStateNotify:withText:, 341
- ticks, 266
- ticks:, 266
- ticksBoth, 266
- ticksBottom, 266
- ticksLeft, 266
- ticksRight, 266
- ticksTop, 266
- timer:, 293, 335
- timerID, 292
- Timers, 61
- title:prompt:default:, 342
- toggle, 282
- toggleWrap, 292
- Toolbar, 6
- Toolbar Reference, 73
- Tools Menu, 381
  - Browse, 383
  - Code Generation, 384
  - Dirty, 385
  - Initialize, 384
  - Inspect Other, 381
  - Inspect Selections, 381
  - Other, 384
  - Outboards, 383
  - Undo Manager, 384
- toolTips:, 250
- Top, 98
- topCorner:, 198, 259
- TopPane, 285–94
- topPaneClass, 331
- topParent, 337
- TrackBar, 128, 264–68
- Transcript Menu, 75
- Transcript Menu
  - About, 78
  - Edit Window, 75
  - Examples, 79
  - Installed Products, 78
  - Manager, 76
    - Add-In Manager, 76

- Bitmap Manager, 76
- Font Manager, 76
- NLS Manager, 76
- Models, 75
- New CompositePane, 75
- New Dialog, 75
- New Window, 75
- Properties, 76
- Templates, 76
- Translator, 76
  - Event Mappings, 77
  - Event Translation, 77
  - Standard Translation, 76
- WindowPolicies, 75
- TreeView, 128, 269–72
- turnedOff, 215
- turnedOn, 215
- turnOff, 215
- turnOn, 215
- turnToPageNumber:, 250
- Typographic Conventions, 3

## —U—

- uncheck, 149, 262
- unchecked, 149, 263
- unchecked:, 158
- uncheckedIndex:, 158
- uncheckIndex:, 156
- uncheckItem:, 156, 339
- Undo, 88, 226, 256
- Undo/Redo List, 88
- Ungroup CompositePane, 81
- unSelectIndex:, 151
- Up, 98, 238
- update, 226
- updateSelection, 139
- updateTab:, 250
- UpDown, 128, 273–75
- upperCase, 173, 180, 226, 256
- Use ClassHierarchy Browser, 117
- Use Fence, 117
- Use Grid, 117
- Use LayoutFrame, 117

Use WBComboBox, 117  
 useImages, 202, 271  
 useImages:, 156, 202, 271  
 User Name, 117  
 usesColor, 357  
 usesFocus, 357  
 usesForeColor, 357  
 usesFraming, 361  
 useTabStops, 156, 193, 209

## —V—

validated, 293  
 value, 149, 156, 162, 168, 173, 180,  
   194, 198, 209, 213, 215, 226, 230,  
   237, 241, 256, 259, 262, 266, 271,  
   274  
 value:, 149, 157, 162, 168, 173, 180,  
   194, 209, 213, 215, 226, 230, 237,  
   241, 256, 259, 262, 266, 274  
 valueIndex, 157, 162, 168, 194, 198,  
   209, 237  
 valueIndex:, 157, 162, 168, 194, 198,  
   209, 237  
 valueIndices, 209  
 valueIndices:, 210  
 vertical, 230, 266, 274  
 verticalScrollBar, 183  
 video, 276  
 video:, 276  
 VideoPane, 128, 276  
 View Menu, 92  
   Create, 92  
   Remove, 92  
   Switch To, 92  
 ViewManager, 9, 10, 61, 75, 80, 83,  
   86, 90, 316, 317  
 visible, 139  
 VisualSmalltalk, 1, 2, 3, 5, 9, 11, 31,  
   38, 52, 77, 93, 196, 257, 304, 312,  
   315, 325  
   compiler, 35, 103  
   obsoletes, 345

## —W—

wantReturn, 173, 180, 226, 256  
 warning:, 341  
 WBAttributeEditor, 364  
 WBAttributeEditor:, 364  
 WBCodeGenerator, 371  
 WBCodeModule, 371  
 WBGraphicObject, 355  
 WBLookPolicy, 361  
 WBMiniBrowserExample, 375  
 WBScrollBarEditor, 364, 365  
 WBStaticGraphic, 128, 277–78  
 WBToolBar, 128, 279–84, 365  
 when:perform:, 139, 346  
 when:send:, 139  
 when:send:to:, 9, 139, 346  
 when:send:to:withArgument:, 140  
 whenValid, 140  
 whenValid:, 140  
 whenValid:with:, 140  
 whenValid:withArguments:, 140  
 whiteFrame, 239  
 whiteRect, 239  
 Widget  
   adding text, 8  
   moving, 8  
   naming, 8  
   placing, 7  
   resizing, 8  
 Widget's Contents, 356  
 Widget Palette, 6  
   Adding icons, 366  
 Widgets, 325  
 Widgets's Initial Size, 356  
 width, 140, 292  
 Window, 285–94  
 WindowBuilder Pro, 1, 5, 13, 51, 76,  
   113  
   Coding in, 51–55  
   customizing, 355  
   Starting, 5  
   Support, 3  
 WindowDialog, 75, 80, 86

WindowPolicy, 45, 48, 75, 83, 96, 292  
WindowPolicy Editor, 45, 49, 83, 95  
windowPolicy:, 292  
windowPolicyClass, 52  
Windows, 330  
windowUnderPoint:, 337  
wizard, 250  
wrap, 226, 256, 274  
wrap:, 226, 256, 274

## —X—

XoteryX, 3, 14, 63, 81, 86

## —Z—

ZipCodeUS, 177  
Zoom Layout, 113  
Z-Order, 18, 23, 111  
    Viewing, 23

## —2—

20Comma10, 176

## —7—

7Comma2, 176  
7Comma4, 176

## Reader Comment Sheet

Name: \_\_\_\_\_

Job title/function: \_\_\_\_\_

Company name: \_\_\_\_\_

Address: \_\_\_\_\_

Telephone number: \_\_\_\_\_ Date: \_\_\_\_\_

How often do you use this manual? ☐ Daily ☐ Weekly ☐ Monthly ☐ Less

How long have you been using this product? ☐ Months ☐ Years

Can you find the information you need? ☐ Yes ☐ No Please comment.

\_\_\_\_\_  
\_\_\_\_\_

Is the information easy to understand? ☐ Yes ☐ No Please comment.

\_\_\_\_\_  
\_\_\_\_\_

Is the information adequate to perform your task? ☐ Yes ☐ No Please comment.

\_\_\_\_\_  
\_\_\_\_\_

General comment: \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

## WE STRIVE FOR QUALITY

To respond, please fax to Larry Fasse at (513) 612-2000.