Cincom

Cincom Smalltalk™

GUI Developer's Guide

P46-0136-06

SIMPLIFICATION THROUGH INNOVATION®

Copyright © 1993–2008 by Cincom Systems, Inc.

All rights reserved.

This product contains copyrighted third-party software.

Part Number: P46-0136-06

Software Release 7.6

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. ParcPlace and VisualWorks are trademarks of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. ObjectLens, ObjectSupport, ParcPlace Smalltalk, Database Connect, DLL & C Connect, and COM Connect are trademarks of Cincom Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

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

VisualWorks 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 © 1993–2008 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

Chapter 3 Controlling the GUI Programmatically

Chapter 5 Menus

Chapter 8 Custom Views

Chapter 9 Configuring Widgets

About This Book

This document is designed to help both new and experienced developers create application programs effectively using the VisualWorks® application frameworks, tools, and facilities.

Audience

This guide assumes that you have at least a beginning familiarity with object-oriented programming. The description of VisualWorks begins at an elementary level, with an overview of the system tools and facilities, and a description of Smalltalk syntax, but does not attempt to be a tutorial.

For additional help, a large number of books and tutorials are available from commercial book sellers and on the world-wide web. In addition, Cincom and some of its partners provide VisualWorks training classes.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.

Example	Description
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File ⟶ New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Smalltalk Tutorial

A new VisualWorks Smalltalk tutorial is available online at:

http://smalltalk.cincom.com/tutorial/index.ssp?content=tutorials

The tutorial information is growing, so revisit this site.

Online Help

VisualWorks includes an online help system. To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select one of the help options.

VisualWorks FAQ

An accumulating set of answers to frequently asked questions about VisualWorks is being compiled in the VisualWorks FAQ, which accompanies this release and is available from the Cincom Smalltalk documentation site.

News Groups

The Smalltalk community is actively present on the internet, and willing to offer helpful advice. A common meeting place is the comp.lang.smalltalk news group. Discussion of VisualWorks and solutions to programming issues are common.

VisualWorks Wiki

A wiki server for VisualWorks is running and can be accessed at:

http://wiki.cs.uiuc.edu:8080/VisualWorks

This is becoming an active place for exchanges of information about VisualWorks. You can ask questions and, in most cases, get a reply in a couple of days.

Commercial Publications

Smalltalk in general, and VisualWorks in particular, is supported by a large library of documents published by major publishing houses. Check your favorite technical bookstore or online book seller.

Examples

There are a number of examples in parcels, installed in the **`examples`** subdirectory, under the VisualWorks install directory. These are referred to frequently in this document, and provide additional help in understanding GUI development in VisualWorks.

1

The VisualWorks GUI Environment

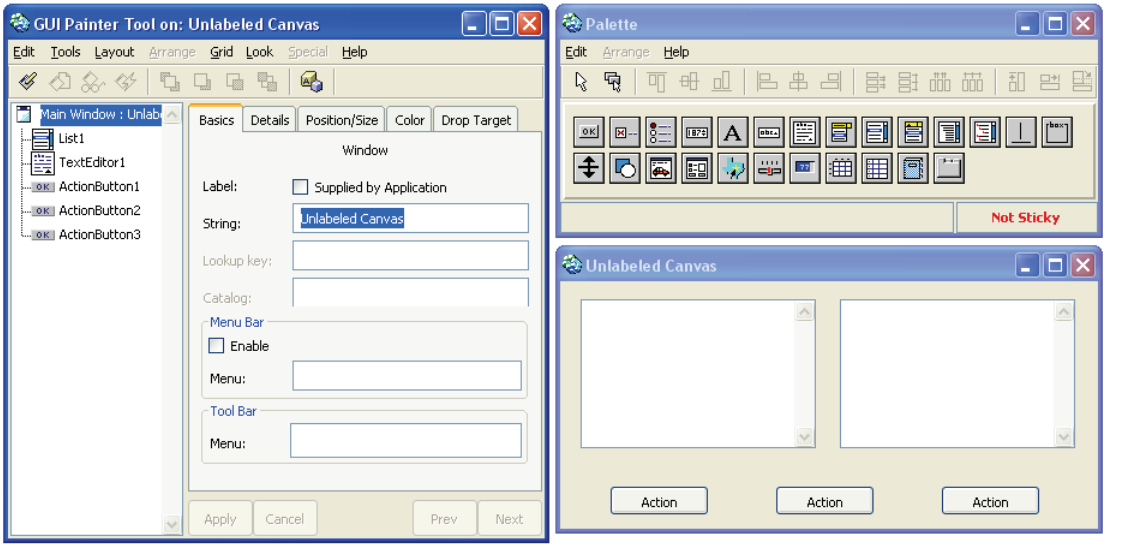
The Graphical User Interface (GUI) building framework is what puts the "Visual" in VisualWorks. Building a GUI is done by visually selecting, placing, and arranging widgets on a "canvas." We call the process "painting," and the tool you use is the UI Painter. The resulting canvas specifies the application window.

Once you have designed the GUI, you connect it to your application's model, so that input to the GUI is passed on to the model, and so that relevant changes in the model are displayed in the GUI. This connecting function is handled by a subclass of ApplicationModel specific to your application. This model is generated by the UI Painter, including stub methods for many essential aspects of the model, according to attributes you specify for the GUI elements.

In this document we address the more mechanical aspects of building and controlling a GUI in VisualWorks. For a lower-level description of how the application framework works in connection with the rest of your application, refer to the "Application Framework" chapter in the *Application Developer's Guide*.

UI Painter

The “Visual” in VisualWorks emphasizes the graphical approach to Graphical User Interface (GUI) design and development. This is provided by the UI Painter.



The UIPainter is initially unloaded from the base image in the commercial version, but is preloaded in the non-commercial version. To load it, open the Parcel Manager, and load the UIPainter parcel on the **Essentials** page.

The Painter tool is in three parts:

- Canvas (lower right) - represents a single window, on which you place widgets, the graphical components of the GUI.
- Palette (top right) - presents a collection of widgets that are commonly used in a GUI, and some widget arrangement buttons.
- GUI Painter Tool (left) - provides a collection of menu commands and buttons for performing formatting and other operations on the canvas, a hierarchical view of the widgets on the current canvas, and the properties of the selected widget.

The Canvas

The Canvas, which starts out blank, is where you place and arrange widgets to “paint” your GUI.

The canvas can be resized to define the size of the resulting window, and settings allow you to specify its initial size when it opens in the application, and to constrain its maximum and minimum sizes.

The widgets you place on the canvas have a number of properties that define their appearance and interaction with your application's data model. Positioning properties determine how the widgets scale or distribute when the window is resized.

Once you have painted a GUI, you "install" it in your application. This writes the canvas definition spec into an interface specification. The specification is used by the application to open the window. You can also edit the specification later by reopening it in the UI Painter.

The Canvas <Operate> menu provides configuration commands that are also available in other parts of the UI Painter, notably those on the GUI Painter Tool.

The Palette

The Palette has one button for each type of widget. Widgets are the elements that you can include in a GUI, including simple elements like labels and buttons, and more complex elements like hierarchical lists and tables.

To add a component to your canvas, such as an input field, you simply click on the **Input Field** icon in the Palette to select it, and then click in the canvas to place the widget. You can then drag the widget around to arrange its position relative to other widgets.

To add several copies of the same widget use the "sticky" widget picker. Then, each time you click on the canvas you drop another copy of that widget, without having to return to the canvas each time. To return to single widget painting, select the "not sticky" picker.

The Palette also has several buttons for doing some basic widget organizing, such as aligning and distributing widgets. These operations are also available on the GUI Painter Tool menus, and in the Canvas <Operate> menu.

GUI Painter Tool

The GUI Painter Tool is "control central" for configuring a canvas and its widgets.

While you may have several canvases open at any given time, there is only one GUI Painter Tool, which is focused on the currently selected canvas. To operate on another canvas, select it to update the GUI Painter Tool.

The GUI Painter Tool has menus and buttons for configuring the canvas and its widgets. These are described throughout this document as the relevant task is discussed.

On the left is a hierarchical Tree View, the components in the current canvas. The canvas itself is included at the top of the list, as **Main Window**. Below it, indented to show inclusion, are any widgets you have placed on the canvas. Each widget has a default name, which you may rename to make more descriptive and useful. Composite parts, which are groups of widgets, are further indented under the composite.

You can select a widget to configure by either clicking on the widget in the canvas or in the widget list in the GUI Painter Tool. The widget list is particularly useful in complex canvases, where a hidden widget may be overlaid with another widget. It also simplifies selecting a nested widget for configuration.

The widget list also shows the tab order between widgets for which tabbing is enabled. Tab order is set from top to bottom. Moving a widget in the list adjusts its place in the tabbing sequence.

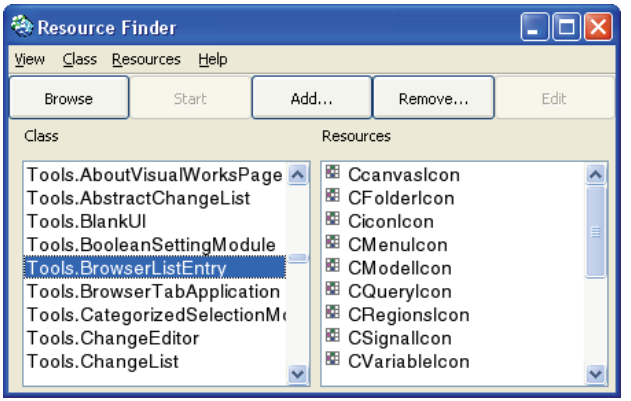
The largest section of the GUI Painter tool is the collection of properties pages. The specific pages and their contents vary from widget to widget, but every widget has **Basics**, **Details**, **Color**, and **Position/Size** pages. Additional pages are added as required by the currently selected widget.

Common properties are described under Setting Properties in Chapter 2, “Building an Application’s GUI.” Additional properties are described individually for each widget in Chapter 9, “Configuring Widgets.”

Once the canvas has been designed, you install it by clicking the Install button, or by selecting the **Edit → Install** menu item.

Resource Finder

Canvases, menus, and graphical images are stored in VisualWorks as *resources*, and saved as specifications in class methods of the Application Model. To find resources once they are created, select **Painter → Resource Finder** in the Launcher to open the Resource Finder.

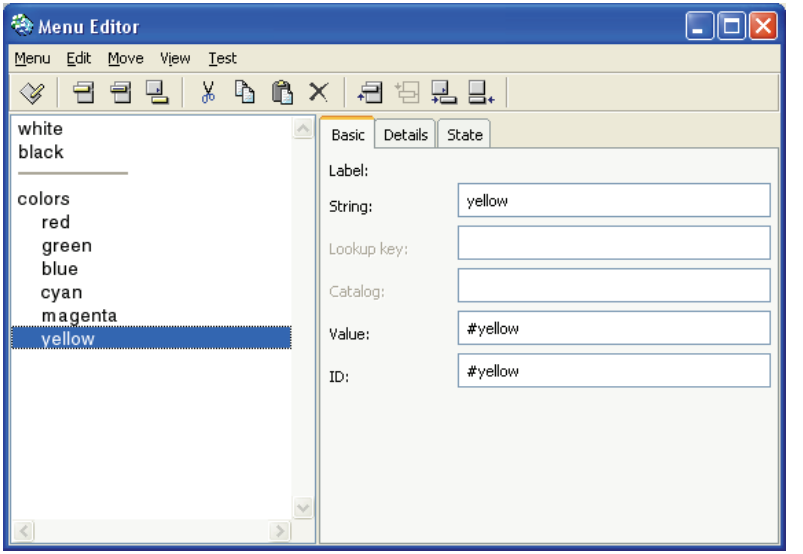


Application classes are listed in the left-hand view. You can filter the list to show tool classes, example classes, and other categories. To do so, use the **View → All Classes** submenu.

An application class can support multiple interfaces, each of which may involve multiple canvases for the VisualWorks main window, secondary windows, and dialogs. The right-hand view lists all of the resource methods for the selected class. Double-click or click **Edit** to open that resource in its editor.

Menu Editor

The Menu Editor provides an intuitive, visual way of building a menu. The menu can be added to the GUI either as a standard, drop-down menu on the menu bar, added to a menu Button widget, or as a pop-up (right-click) menu to the window area. To open the Menu Editor, select **Tools → Menu Editor** in the GUI Painter Tool or **Painter → Menu Editor** in the Launcher.



Menu commands and buttons are provided for adding an item, adding a sub-item, and for rearranging items. Sub-items are accessed as sub-menus off of the parent menu.

Menu levels are indicated by indenting. To change the level of an item, select it and click on the left or right arrow button. To change the order of an item, select it and click the up or down buttons.

There are also three properties pages for each menu item. The **Basic** page specifies the displayed String menu label, the lookup key and catalog, if used, the item's return value, and its ID, for programmatically accessing the item. The **Details** page specifies any modifier keys needed for the item, the icon for the item, if any, and its help text. The **State** page sets the item's initial state.

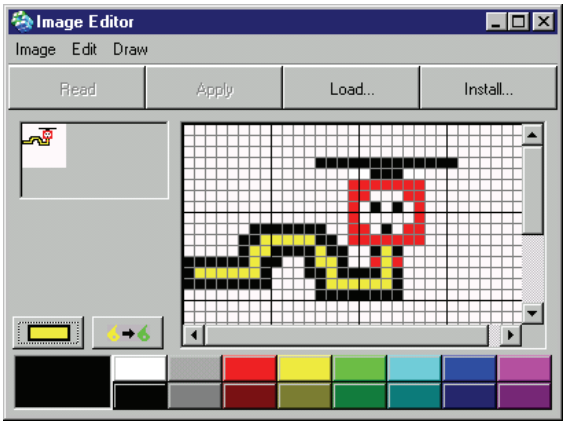
An application frequently needs to modify a menu or the state of its items while running, either to indicate state or to activate and deactivate items. Details on how to create menus and control them during runtime are provided in Chapter 5, "Menus."

Image Editor

Your GUI is often enhanced by the inclusion of icons, simple graphics that indicate a widget's function. Menu items and buttons, for example, are often clearer with an icon to suggest visually what they do.

VisualWorks includes an Image Editor that you can use to draw an image pixel by pixel, and then store it in a resource method. The Image Editor is best suited for producing small images, such as icons and cursor shapes.

To open an Image Editor, choose **Tools → Image Editor** in the GUI Painter Tool or **Painter → Image Editor** in the VisualWorks main window.

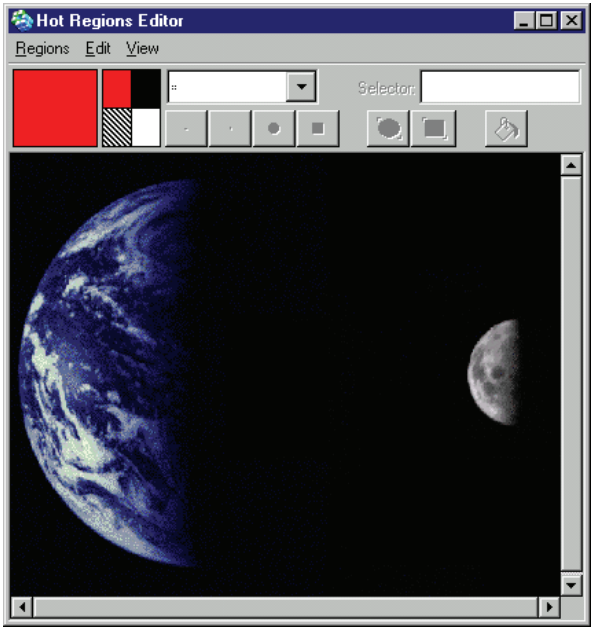


Paint the desired image in the scrollable pixel grid. The controls are pretty standard for simple paint programs.

To make the graphic available to your application, click the **Install** button, then specify your application class as the class into which to install the graphic, and a method name for the graphic. This installs the graphic as a resource, which you can access in the resources browser. The method is installed as a class method in a resource protocol of a selected class.

For more about graphics in VisualWorks, refer to "Working With Graphics and Colors" in the *Application Developer's Guide*. Instructions for incorporating a graphic in your application's GUI are provided at relevant points throughout the rest of this document.

Hot Region Editor



Clickable images can give a very modern look to an otherwise dull GUI. The Click Map widget allows you to add one to your canvase. The Hot Regions Editor allows you to define clickable regions.

To define hot regions, you load a graphic resource into the Hot Regions Editor as a back drop. Then, using drawing tools, draw and paint areas to be included in the region, and specify a message selector to be sent when the region is clicked.

For detailed instructions, refer to “Click Map” in Chapter 9, “Configuring Widgets.”

2

Building an Application's GUI

Overview

This chapter provides an overview of how to build a graphical user interface (GUI) and link it to the domain model(s) of a VisualWorks application.

Separating Domain and Application Models

The main guiding principle in developing a VisualWorks application is to clearly and cleanly separate the *domain* model from the *application*, or GUI, model.

The *domain model* is one or more objects that represents entities in the application's domain. These may represent, for example, business objects such as customers, inventories, and processes. In a simple application, the entire domain might be represented by a single class, but usually will involve several.

The *application model*, on the other hand, represents the presentation of the domain to the user, and so includes the GUI and its support mechanisms. This maintains one or more windows and their widgets, and any logic needed for presenting data from the domain.

In general, the domain model should remain free of user-interface code. Any instance variables or methods that are necessary purely to support the mechanics of the user interface belong in the application model. This separation of responsibilities makes it easier to reuse your domain models with other interfaces.

While the split between domain and application models is not always completely clear, the distinction often helps when deciding where to locate variables and methods.

A complete GUI application consists of its application and domain models. In this chapter we address how to build the GUI. Connecting the two models, while maintaining their separation, involves using value models, and is discussed in Chapter 4, "Adapting Domain Models to Widgets."

GUI Development

While there must be a domain model to complete the linking, a great deal of preliminary work can be begun by working on the GUI, so it is not necessary to develop the domain model before building the GUI. A fairly common development approach in object-oriented programming is to develop, or at least prototype, the GUI before completing the domain model.

The VisualWorks Painter tool both creates the UI specification and the application model, so this stage goes a long way to beginning application development.

In this chapter, we begin with an overview of the basic operations of building the GUI using the UI Painter tool. From there we move to a more complete description of building an application view for a domain. Finally, several approaches to and options for building the adaptors that connect the GUI and domain are described.

Loading the UI Painter

The UI Painter is a loadable component in VisualWorks. Before you can begin painting a canvas, you need to load it.

Load the UIPainter parcel using the Parcel Manager, **System → Parcel Manager**, then select and load the **UIPainter** parcel, on the **Essentials** page.

Creating a Graphical User Interface

To create the visual portion of a graphical user interface, you specify the contents and layout of each window in your application. These windows include the application's main window, secondary windows, and dialog boxes.

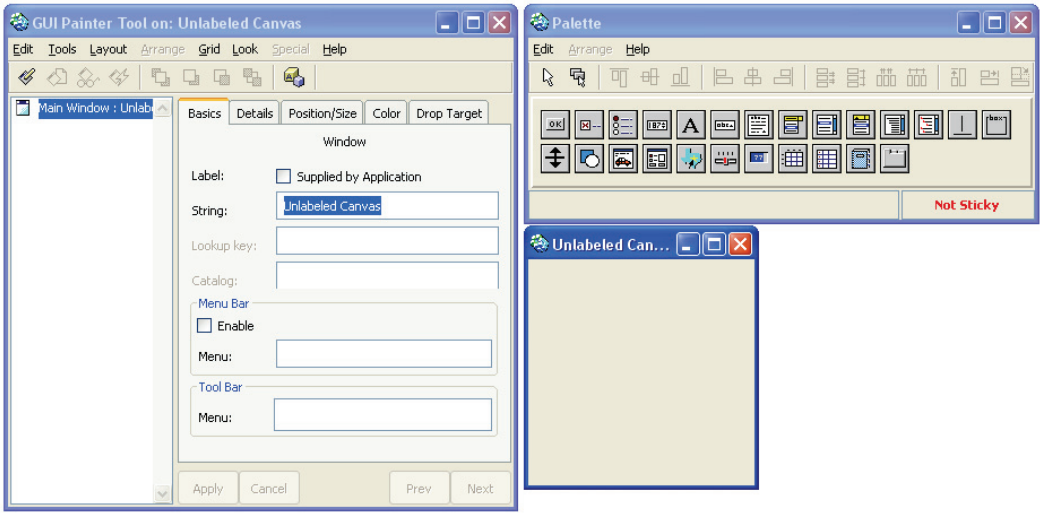
Part of the procedure generates stub methods and value holders to link up with the domain model. Later, you program application-specific behavior for each of the windows you specified. And, as an alternative

mechanism for linking the domain and application models, you can use the Trigger-Event system, as described in "Configuring Dependencies Using Events" in Chapter 4, "Adapting Domain Models to Widgets."

"Painting" a Window

You create windows by creating a visual specification using the UI Painter. The painter tool provides a canvas, on which you arrange widgets, a palette containing widgets, and a canvas tool, which gives quick access to a number of common UI layout commands.

To open the Painter, click on the **New Canvas** button, or select **Painter → New Canvas**, in the Visual Launcher.



The UI Painter opens with three windows: the GUI Painter Tool (left), the widget Palette (top right), and the canvas (bottom right).

The canvas is a work area in which you add and arrange visual components until it looks like the window you want in your application. If your application uses several windows, you create a separate canvas for each window in the application. One window will be the main window, and the rest will be sub-windows.

You paint the blank canvas by placing and arranging widgets on it. To place a widget, click on a widget in the palette, then move the mouse to the canvas and click once. A representation of the widget is displayed at the cursor. Move it where you want the widget, then click again to drop the widget.

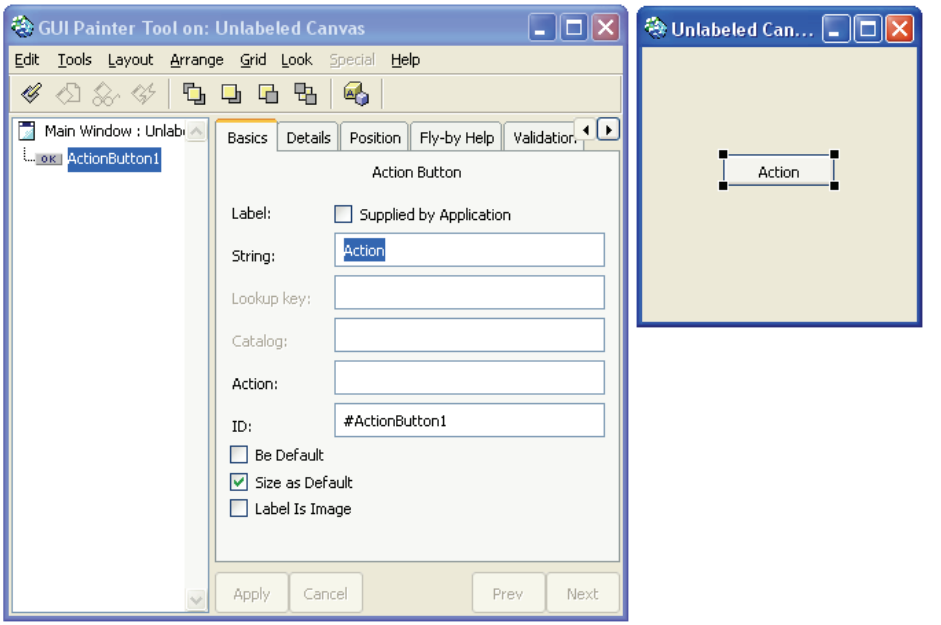
If you want to place several of the same widget on the canvas, select the Sticky Select button before selecting the widget. Every time you drop one widget, you can drop another, until you select another widget or reselect the single-place button.

Setting Properties

Every window and widget has an associated collection of properties, grouped into several pages. A few properties are common to all windows and widgets, but most vary from object to object.

Properties define a variety of visual attributes, such as font, color, borders, and so on. For some widgets, such as input fields, properties also indicate the nature of the data to be displayed and how that data is to be referenced by the application.

Properties are initially set using the properties pages in the GUI Painter Tool while you are constructing the GUI.



Most of the properties available for setting are self-explanatory. If you want to change some characteristic of a window or widget, look in the property pages, and you'll usually find the property setting you need. Many of the properties are described in other sections of this manual, either in discussions of general features or of the individual widgets (refer to Chapter 9, "Configuring Widgets").

A few fields need some initial explanation:

String
A string for the widget label. The label may be supplied programmatically or by a graphic in many cases, which you indicate by marking a check box for the option, and the string field is grayed-out.

Message
The name of the class method in the application model providing the label, if the label is provided by the application (**Supplied by Application** checkbox). The label can be either a string or by a graphic (**Label is Image** checkbox).

Aspect
The name of the property in the application model that represents the value model for the widget. An instance variable and stub accessor method for the aspect are created when you select **Define** for this widget. The value model is typically a ValueHolder, if the widget reflects a property within the application model, or an AspectAdaptor, if the widget reflects a property in a separate domain model. An Aspect Path may also be specified in this field (see "Defining Adaptors in the UI Painter" in Chapter 4).

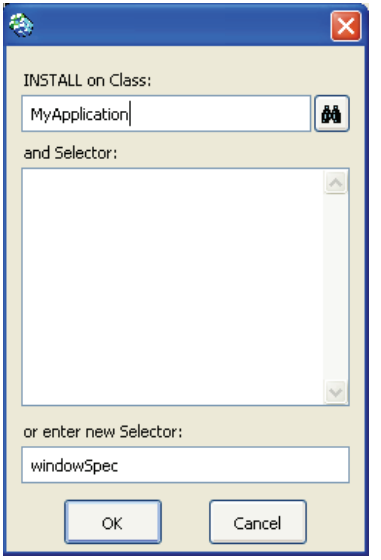
ID
An identifier for the widget. A default identifier is provided for each widget, so you can easily identify it in the list of widgets. This identifier is used to access this widget programmatically, and to register event handlers with the widget. You may change the default name to something more descriptive or keep the default name.

Catalog and Lookup Key
These fields specify a message catalog and lookup key into the catalog. This fields are displayed for widgets that take a label, such as the label, action button, and radio button, and for the **Fly-by Help** for widgets. The fields are activated when the **Show UI for internationalization** option is activated on the **Tools** page of the Settings Tool. For additional information about message catalogs and their use, refer to the VisualWorks *Internationalization Guide*.

Installing the Canvas

At any time in the painting process, you can save the canvas by *installing* it in an application model. Installing the canvas creates an *interface specification*. The specification is stored as a class method in the application model.

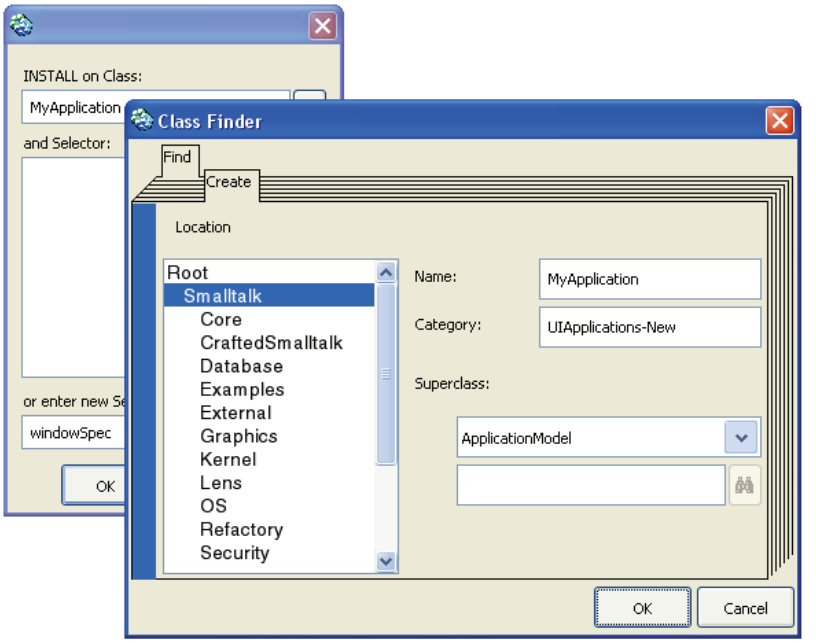
To install a canvas, either click the **Install** button in the GUI Painter Tool, or select **Install...** in the <Operate> menu for the canvas. The Install dialog allows you to enter the necessary information.



For a new canvas, you need to enter a class name in the **Install on Class** field. This can be a new class name or the name of an existing class. For a new class, you will be asked for the appropriate superclass in a subsequent dialog.

You must also either enter or select a method selector (message name) for the specification. The default is windowSpec, which is the standard name for the main window specification. If you are defining a secondary window, you should pick another name. Click **OK** to save the specification.

If you specify a class that does not already exist, you are asked to select a superclass and a name space for it. For an application, you typically select **ApplicationModel** as the superclass. The name space is typically either one created for your application or **Smalltalk** (refer to "Name Spaces," Chapter 6 in the *Application Developer's Guide*, for more information).



The ApplicationModel superclass provides support functions for GUI-oriented applications, so it is usually the right choice for an application. SimpleDialog provides support features for dialogs (see Chapter 7, "Dialogs" for more information). On the other hand, if the class must be a subclass of some other class, select **Other** and enter its name.

The **Category** field sets the category for the new class. You can use an existing category or a new category name that better identifies your work.

If you have Store installed, after closing this dialog you will be prompted for a package. Select a package or "none" and continue.

Reopening a Canvas

When you save the canvas, its specification is added to the *resources* for the application. After you've closed the canvas, you can open it again for editing using the Resource Browser (**Painter → Resource Finder** menu item in the Visual Launcher). In the Resource Browser, select the class name containing the canvas, then select the specification for the canvas and click **Edit**. You can browse the specification method, and evaluate the code in its method comment, to open the specification in its editor.

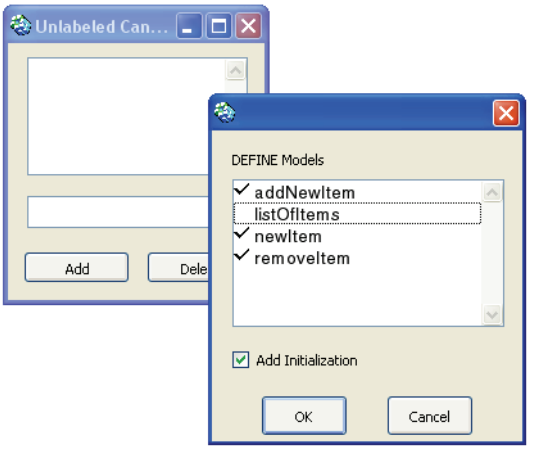
To save any changes you make to a canvas, you must reinstall it.

Defining Value Models

Once a canvas is installed, you can define instance variables and stub accessor methods for the aspects and actions you identify for each widget. For most widgets, the names of these variables and methods are specified in the **Aspect** property. The Action Button widget has only a method (no instance variable), and its name is specified in the **Action** property field. Some widgets do not use variables or methods at all, and so do not have a corresponding property.

To use the Trigger-Event mechanism, do not specify the **Aspect** or **Action**. These are handled differently, as described below in "Configuring Dependencies Using Events". You must specify an **ID** to use Trigger-Events.

Once you have entered aspect names for one or more widgets, select the widgets, either individually, multiply, or by selelcting the canvas itself to define all. Then click the **Define** button in the GUI Painter Tool, or select **Define** in the <Operate> menu for the canvas. A confirmation dialog listing the names of aspects and actions for the selected widgets is displayed.



In the list, each item that will be defined is marked with a check mark. To remove the check mark, and so skip that method, click on the item.

Unchecking an item is important in subsequent define operations, because redefining will overwrite any custom changes you make to the method.

The **Add Initialization** checkbox determines whether the stub method will include initialization code. Without initialization, the instance variable for the aspect is simply returned. With initialization, the method includes lazy initialization code, testing for a value and, if not set, assigns an initial value. The type of value assigned depends on the widget.

Once the stub methods are generated, you can edit them to provide the specific processing your application needs. For example, to connect the UI to its domain model, you must redefine some aspect methods to use aspect adaptors. Refer to Chapter 4, "Adapting Domain Models to Widgets" for more information.

Testing the User Interface

When you have installed the window's canvas in an application model, you have a minimal application that can be started. At this point, the widgets in the window exhibit generic behavior, because you've not added any specific behavior to exhibit or data to display. You can, however, see how the interface will look under running conditions.

To start the application, with the canvas open, click the **Open** button in the Canvas Tool, or select **Open** in the canvas <Operate> menu. Move the new window outline to a location on the screen and click.

You can also use this method for starting the application later, as you add specific behavior to the application.

Formatting a Canvas

The following sections describe operations for formatting and arranging widgets, to adjust how the window will appear in the application. Most of the operations are performed in the canvas while assembling the GUI.

Some formatting can be set or changed programmatically, while the application is running. Instructions are provided for doing this along with the canvas instructions. For general instructions on accessing widgets from a running application using the widget's ID, see Chapter 3, "Controlling the GUI Programmatically."

Setting the Window Size

While editing a canvas, you change the size of the window by dragging the corners of the canvas to any desired size. Generally you select a size and proportion that accommodates any widgets you use in the GUI. As you design the canvas, you will likely change the size.

Once you have finished laying out the GUI, you need to ensure that the window opens at the size you designed. To do this, select the **Main Window** item in the GUI Painter Tool's widget list, and go to the **Position/Size** page.

At the bottom of are three sets of height and width dimensions. The **Specified** dimensions give the default opening size. To set this to the dimensions that you have sized the canvas, simply click the **Specified** button, and the current dimensions are filled in. You may enter other dimensions if you prefer.

It is also frequently useful to constrain a window to a certain size, so that a user cannot change its size to larger or smaller than certain dimensions. This is useful when the interface becomes unusable outside of a range of values. These dimensions are provided by the **Minimum** and **Maximum** dimensions. You can enter precise dimensions, but it is easier to resize the canvas to the smallest usable size and click the **Minimum** button. Similarly, resize the canvas to the largest desireable size, and click the **Maximum** button.

If you allow the window to be resized, you will also want to decide how widgets are sized. Refer to "Sizing a Widget" for directions on controlling widget sizes.

Setting the Window Opening Position

In the GUI Painter Tool for a canvas, you can specify how the window will be placed when the application is opened. Go to the **Position/Size** page, and select an opening option for your application window.

In the top group, the options are:

System Default: Opens the window as specified as the default in the Settings Tool, which is either user placement or automatic placement.

User Placement: Opens the window by user placement, regardless of the default.

Advanced: Activates further options (below).

If you select **Advanced**, the further options are:

System Default: Opens the window as specified as the default in the Settings Tool.

Screen Center: Opens the window centered on the screen.

Mouse Center: Opens the window centered on the mouse.

Last/Saved Position: If **Auto** is checked, the window will open where it was last closed. If **Auto** is unchecked, the window will open each time in the same location.

Cascade: Each time the window opens below and to the right of where it last opened. This is useful especially if several copies of the window can open in succession.

Specified Position: Opens with the top left corner at the specified screen coordinate.

After selecting the option, **Apply** the change and **Install** the canvas.

Adding Scrollbars to a Window

Scrollbars on a window allow a GUI to contain widgets that might not always be displayed, depending on window size. If this is a possibility, you can add horizontal and/or vertical scrollbars to your window.

1. In the window's canvas, make sure no widget is selected, or select the **Main Window** item in the widgets list.
2. On the **Details** properties page, turn on the desired scroll bars.
3. Apply the properties and install the canvas.

Adding a Menu Bar

To add a menu to a window, you create the menu using the Menu Editor (refer to Chapter 5, "Menus") and enable the menu in the canvas.

1. In the canvas for the window, make sure no widget is selected, or select the **Main Window** item in the widgets list.
2. On the **Basics** properties page, turn on the **Enable** switch for the **Menu Bar** property.
3. In the **Menu** field, enter the name of the menu-creation method, defined for the menu in the Menu Editor.
4. Install the canvas.

You can use the Menu Editor to create the menu either before or after enabling the canvas. Each first-level entry in the menu appears in the menu bar, but *only if* it has a submenu.

Adding Fly-by Help

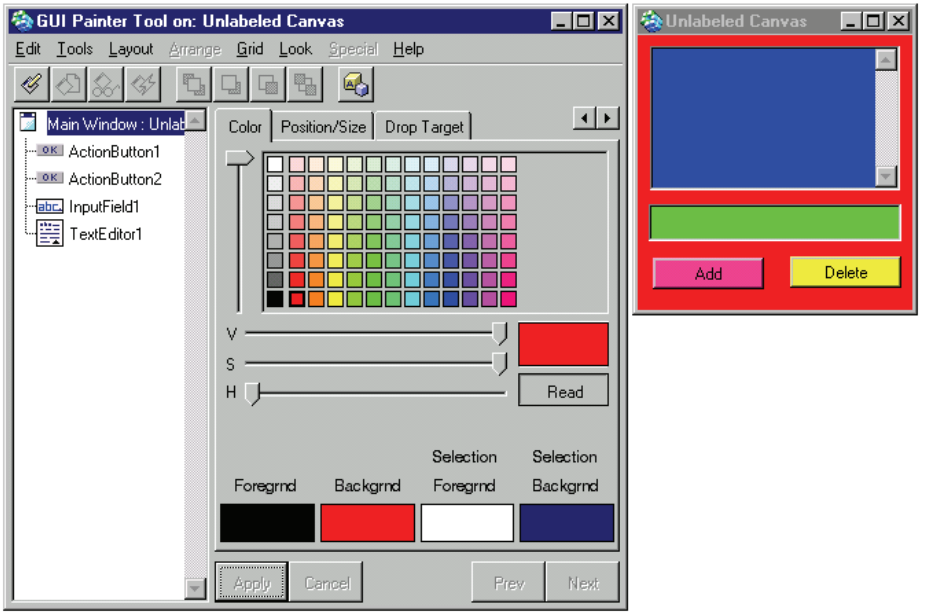
The **Fly-by Help** page allows you to specify the String that displays when the curser pauses over a widget.

The **Default** field simply takes a String value, which is used unless a **Catalog** and **Lookup key** are specified and found. Message catalogs are part of the internationalization features, and are described in the *Internationalization Guide*.

If you check **Supplied by Application**, the **Message** field becomes active. Enter the method selector for the class method that returns the help text.

Setting the UI Colors

Setting colors for a window and for widgets within the window use the same property dialog, and in fact work together.



Caution: The color selections represented above are among the absolute worst in GUI design. Such color choices will evoke at least disgust, and possibly violent illness, in your users. *DO NOT* use such colors in your application.

Windows and widgets have four color zones:

- Foreground
- Background
- Selection foreground

- Selection background

On the **Color** properties page, you can apply a color to any of these zones.

By default, each zone is set to **none**, which means that the window or widget inherits its colors. A window inherits the platform window manager's colors, and widgets inherit from either a containing widget or the window. Unless you specifically want to highlight some feature of the GUI, you should leave all color selections at their default.

When you do select colors, be sensitive to good taste (unlike the above example). Also, be aware that some forms of color blindness make certain combinations useless. For example, many people cannot distinguish between red and green, so a red foreground on a green background ought to be avoided.

To change a color:

1 In a canvas, select the window or widget whose color you want to set.

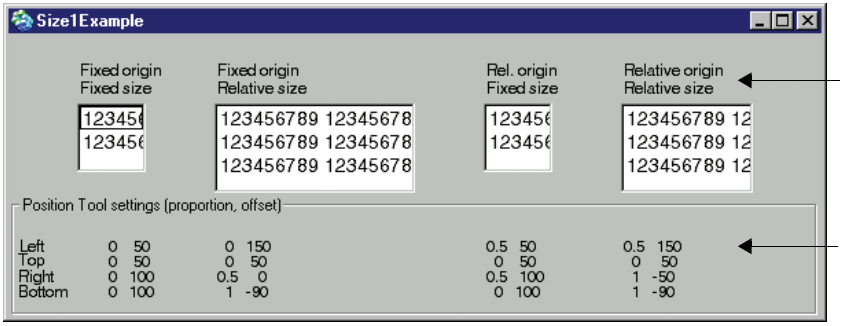
2 On the **Color** properties page, select the desired color from the color chart.

3 Click on the color zone on the **Color** page.

To revert to none, click on the zone a second time with the same color selected.

4 Apply the properties and install the canvas.

Sizing a Widget



These lists . . .

. . . are sized and positioned by these settings in a Position Tool

In the canvas, you can set a widget's size by dragging its selection handles, stretching or shrinking the widget's dimensions.

You can make the widgets in a multiple selection all equal in height, width, or both, using the **Arrange → Equalize…** command in the GUI Painter Tool menu. There are also buttons on the Palette for doing these operations.

Widgets appear in their painted size when the window is opened. When the window size is fixed, nothing more normally needs to be done. However, when the window's size is variable, you may want to arrange for the widget to adjust its size in relation to that of the window. You can use the **Layout → Relative** command on the Canvas Tool to arrange for automatic resizing in both the vertical and the horizontal dimensions.

For more complicated situations, or for more precise control, you can set properties on the **Position** properties page, as described in the following sections.

There are three layout schemes for widgets, which provide different ways to specify the widget's size relative to a point:

Bounded - This layout sets the widget position by specifying the left, top, right, and bottom location, each as an offset from a proportional position within the window. The widget's size may vary as the window is resized.

Unbounded - This layout is available only for Action Buttons, Checkboxes, Radio Buttons, and Labels, which are naturally variable in size based on the size of their label. When set to unbounded, the widget is positioned by setting the widget's origin (top left corner), and its position relative to the origin. Size is determined by the label size.

Origin + Width and Height - This layout sets the widget's origin (top left corner), and its size as a fixed height and width.

Making a Widget's Size Fixed

To make a widget's size constant, use the Origin + Width and Height layouts. (You can use Bounded layout, but it is more difficult.) Set the widget's origin the Left (**X**) and Top (**Y**) property settings, and then set its height and width in pixels. A fixed size is commonly used for buttons and labels.

1. In a canvas, select the widget whose size is to be fixed.
2. On the **Position** properties page, click the **Origin + Width and Height** layout button.
3. Set the left (X) proportion and offset, and the top proportion and offset.
4. Set the **Height** and **Width** sizes in pixels.

5 Apply the properties and install the canvas.

Making a Widget's Size Relative

You can cause a widget to expand or shrink in concert with the window by setting its Right Proportion to be different from the Left Proportion, or by setting the Bottom Proportion to be different from the Top Proportion. This is especially useful for widgets that can use additional space, such as text editors, lists, and tables. Input fields are often made relative in the horizontal dimension only.

Online example: Size1Example

1 In a canvas, select the widget whose size is to be relative.

2 On the **Position** properties page, set the Right Proportion to a value that is larger than the Left Proportion. (A right proportion of **0.5** keeps the right edge anchored at the window's midline while the left edge is anchored to the window's left edge.)

3 Set the Right Offset to the distance you want between the widget's right edge and the imaginary line identified by the Right Proportion.

4 Set the Bottom Proportion to a value that is larger than the Top Proportion.

5 Set the Bottom Offset to the distance between the widget's bottom edge and the imaginary line representing the Bottom Proportion.

6 Apply the properties and install the canvas.

Applying Explicit Boundaries to an Unbounded Widget

Four widgets are naturally variable in size: labels, action buttons, radio buttons, and check boxes. These widgets change in size to accommodate their textual labels, which expand and shrink on different platforms because of font differences. Unlike most widgets, which have four boundaries, the variable-size widgets are said to be *unbounded*.

Sometimes it is preferable to convert an unbounded widget so it is bounded like other widgets. As shown in Size2Example, the advantage is that you can make a series of buttons have equal dimensions, for example. There is a slight hazard in converting an unbounded widget, however; on a different platform, a font change in the widget's label may cause the label to expand beyond the widget's unyielding boundaries.

Online example: Size2Example

1 In a canvas, select an unbounded widget such as a label.

2 In the Canvas Tool, select **Layout → Be Bounded**.

Alternatively, select the Bounded button  on the **Position** page.

3 Apply the properties, if necessary, and install the canvas.

To reverse the operation, select **Layout → Be Unbounded**.

Positioning a Widget

The basic way to set a widget's position is by dragging it to the desired position in the canvas. This determines the widget's initial position relative to the window's upper left corner.

In the GUI Painter Tool, use the commands on the **Grid** menu to turn on/off and configure the grid on the canvas. When turned on, widgets snap to position relative to the grid, which simplfies the task of aligning widgets.

Widgets appear in their painted position when the window is opened. When the window size is fixed, nothing more normally needs to be done. However, when the window's size is variable, you may need to arrange for the widget to adjust its position relative to the size of the window. You can use the **Layout → Relative** command on the GUI Painter Tool to arrange for automatic repositioning in both the vertical and the horizontal dimensions.

For more precise control, several options are provided on the **Position** properties page.

Making a Widget's Origin Fixed

Making a widget fixed is useful when the window's size is fixed. When the window's size is variable, this approach works best for a button or other fixed-size widget that is located along the left or top edges of the window.

Online example: Size1Example (start it and then resize the window to see the effect)

1 In a canvas, select the widget whose position is to be fixed.

2 On the **Position** properties page, set the **Left** and **Top Proportions** to 0. These proportions control whether a widget moves relative to the window size. Setting these properties to 0 causes the widget's origin to remain fixed in place.

3 Set the **Left Offset** to the desired distance between the window's left edge and the widget's left edge (in the example, **50** pixels).

4 Set the **Top Offset** to the desired distance between the window's top edge and the widget's top edge (**50**).

5 Apply the properties and install the canvas.

Making a Widget's Origin Relative

A relative origin causes the widget to move farther away from the left and top edges of the window when the window grows and closer when the window shrinks. This is useful for keeping an object centered in the window and for shifting one widget that is placed below or to the right of another widget that expands and shrinks in size.

You can also make the origin relative in only one dimension. In the example, the origin shifts horizontally as the window is resized, but it maintains a stable offset from the window's top edge.

Online example: Size1Example

1 In a canvas, select the widget whose position is to be relative.

2 On the **Position** properties page, set the **Left Proportion** to the fraction of the window's width from which the **Left Offset** is to be measured. (In the example, a left proportion of **0.5** causes the widget to remain anchored at the window's midline.)

3 Set the **Left Offset** to the distance you want between the widget's left edge and the imaginary line identified by the **Left Proportion** (**50** pixels).

4 Set the **Top Proportion** to the fraction of the window's height from which the **Top Offset** is to be measured. (In the example, a top proportion of **0** anchors the widget's top edge at the top edge of the window, which is the same as keeping the origin fixed in the vertical dimension.)

5 Set the **Top Offset** to the distance you want between the widget's top edge and the imaginary line identified by the **Top Proportion** (**50** pixels).

6 Apply the properties and install the canvas.

Giving an Unbounded Widget a Fixed Position

An unbounded widget has no left, right, top, and bottom sides because its boundaries are not fixed. However, it does have a reference point that can be positioned in either a fixed or relative location in the window. By default, the reference point is the origin of the widget (the top-left corner).

1 In a canvas, select an unbounded widget such as a label.

2 On the **Position** properties page, set all of the proportions to 0.

3 Set the **X** and **Y** off sets to the coordinates of the widget's top-left corner relative to the top-left corner of the window.

4 Apply the properties and install the canvas.

Giving an Unbounded Widget a Relative Position

Online example: Size2Example

1 In a canvas, select an unbounded widget (in the example, select one of the unbounded buttons on the left to examine its properties).

2 In the **Position** properties page, set the **X** Proportion to the fraction of the widget's width at which the reference point is to be positioned (**0.5**).

3 Set the **Y** Proportion to the fraction of the widget's height at which the reference point is to be positioned (**0**).

4 Set the **X** Proportion to the fraction of the window's width at which the widget's reference point is to be anchored. (In the example, an **X** Proportion of **0.25** keeps the widget's reference point anchored one-fourth of the way across the window.)

5 Set the **X** Offset to the distance you want between the widget's reference point and the imaginary line identified by the **X** proportion (**0**).

6 Set the **Y** Offset to the fraction of the window's height at which the widget's reference point is to be anchored. (In the example, a **Y** proportion of **0** keeps the widget's reference point a fixed distance from the window's top edge.)

7 Apply the properties and install the canvas.

Grouping Widgets

Windows frequently have clusters of widgets that work closely together. Such clusters are usefully dealt with, for formatting purposes, as a single object. For this purpose, the widgets can be formed into a group.

Once grouped, the whole group can be moved on the canvas, retaining its arrangement.

Making a Group of Widgets

To make a group out of several widgets:

1 Select the widgets to be grouped, by selecting one widget and then hold down the <Shift> key while selecting additional widgets.

2 Select **Arrange → Group** either in the <Operate> menu or in the GUI Painter Tool.

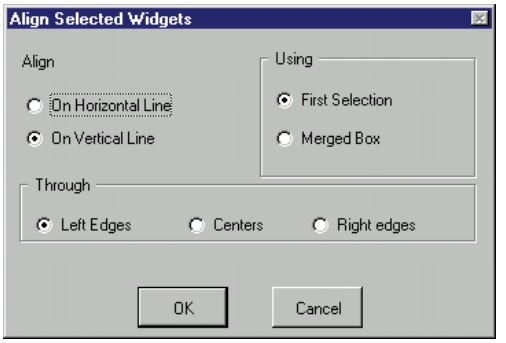
In the widgets list in the GUI Painter Tool, notice that a new pseudo-widget, Composite1 (or some higher increment if there are already composites) is added to the list. The component widgets are listed under it and indented. The list can be collapsed or expanded as desired, to hide or show the component widgets.

To ungroup a group of widgets, select **Arrange → Ungroup**.

Editing Widgets in Groups

You can edit an individual widget within a group, without ungrouping. To do so, in the hierarchical widget list in the GUI Painter Tool, select the desired widget within the group. Then, edit its properties as usual.

Aligning Widgets



When painting a canvas, you frequently need to make several widgets align along a vertical or horizontal line to give a neat, regular appearance. For precise control you can use the Align dialog, shown above and described below. There are also a set of alignment buttons on the Palette, giving quick access to the alignment functions. You can also use the **Position** properties page to precisely position widgets, but would usually choose to do so only due to scaling consideration

1 In a canvas, select the widgets to be aligned.

2 In the GUI Painter Tool or the Canvas <Operate> menu, select the **Arrange → Align** command.

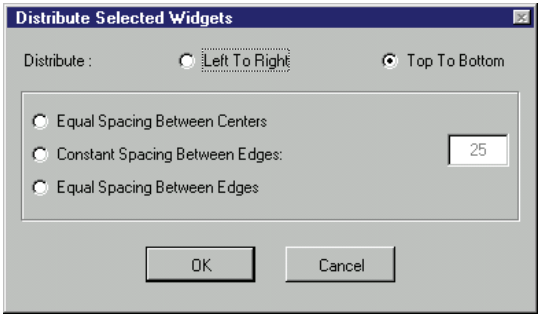
3 In the Align dialog, select **on horizontal line** when aligning side-by-side widgets. When aligning widgets in a column, select **on vertical line**.

4 In the Align dialog, select **first selection** when the widgets are to be aligned with the first widget that was selected. Select **merged box** to align the widgets on a line halfway between the two most extreme positions within the group of widgets.

5 In the Align dialog, select the edges, or the centers, to be aligned.

6 Install the canvas.

Distributing Widgets



When painting a canvas, you frequently need to make the spaces between several widgets equal. For precise control you can use the Distribute dialog, shown above. You can also use the distribution buttons on the Palette.

1 In a canvas, select the widgets to be spaced.

2 In the GUI Painter Tool or Canvas <Operate> menu, select the **Arrange → Distribute** command.

3 In the Distribute dialog, select **left to right** for widgets that are to be spaced in a horizontal row. Select **top to bottom** for columnar distribution.

4 In the Distribute dialog, select the type of spacing. For **constant spacing between edges**, you must specify the number of pixels to place between each pair of widgets.

5 Install the canvas.

Changing a Widget's Font

When a widget's default font is not suitable, you can use the **Font** menu in the widget's properties to choose an alternative font. The built-in fonts are:

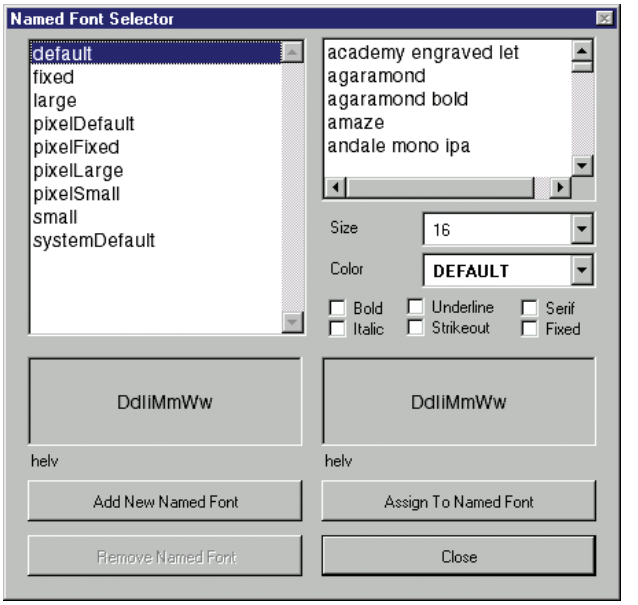
- **System**, for a font that matches the current platform's system font, when available. The font varies depending on platform and Look policy. (Defined in the Look Policy classes.)
- **Default**, for the default font for the widget. The font varies with the Look Policy and the widget.
- **Pixel**, Small, Medium, Large and Fixed defined in TextAttributes.
- **Standard**, Small, Medium, Large and Fixed defined in VariableSizeTextAttributes.

The drop-down lists under **Pixel** and **Standard** also list any Named Fonts you have defined.

Named Fonts

VisualWorks does not automatically add all fonts on your system to the fonts list. However, you can selectively add these fonts and assign them a name for use by name. Once defined, a named font is added to the fonts list in the GUI Painter and other utilities.

Open the Named Fonts tool from the GUI Painter Tool, either by clicking the **Define Named Fonts** button or selecting **Edit → Define Named Fonts**.



To add a named font:

1. Click **Add New Named Font**, and enter a descriptive name in the prompter.
2. With the new name selected, assign attributes to it by selecting the font (as known to the system) in the drop-down list, size, color, and other characteristics.
3. Click **Assign To Named Font**, and **Close** the tool.

The named font can now be assigned to a text widget or field by selecting it on the **Details** page.

The named font definitions are saved in the image the next time you save it. You may also either file out the definitions using **Edit → Named Fonts → File Out…** in the GUI Painter Tool.

More importantly, you can install the named fonts into your application by selecting **Edit → Named Fonts → Install In Application…** in the GUI Painter Tool. This option adds a definedNamedFonts instance method to your application model. Note that this option is only available when the current application has already been installed.

Changing the Tabbing Order

When an application is running, users can use the <Tab> key to shift the keyboard focus from one widget to the next in a window, without having to move the mouse.

The <Tab> key moves focus to each widget on the *tab chain*. You add a widget to the tab chain by turning on its **Can Tab** property on the **Details** property page. Passive widgets such as labels and dividers do not have a **Can Tab** property, so they cannot be put on the tab chain.

In a text editor widget, you should either turn off the **Can Tab** property in a text editor or set the **Tab Requires Control Key** option, if you want the editor to interpret the <Tab> key as a literal character to be entered into the text. Otherwise, <Tab> will advance to the next widget. With **Can Tab** inactive, the user cannot tab out of the text editor. With **Tab Requires Control Key** set, <Ctrl>+<Tab> tabs out of the text editor.

The order in which the <Tab> key advances the focus is the order in which the widgets are listed in the GUI Painter Tool widget list, with the widget at the top of the list being the first in the tab chain. To change the tab order, select a widget and then click one of the ordering buttons:

Bring to Front (last tab position)

Bring Forward (backward one tab position)

Send Backward (forward one tab position)

Send to Back (first tab position)

Tab order (first to last) is in reverse of the widget Z-order (front to back). So the front most widget is last in the tab chain.

For composite parts, arrange the tabbing order within the group. Then, move the group to the appropriate position.

Opening and Closing Windows

The usual way of opening an interface window for an application is to ask an application model to open one of its interface specifications. An application is frequently started by opening its main window in this way.

Opening the Main Window

When the application's main window spec name is #windowSpec, you can just send open to the application model:

```
Editor1Example open
```

If the main window's spec is named other than #windowSpec, send openWithSpec: to the application class.

```
Editor1Example openWithSpec: #windowSpec
```

These expressions can either be incorporated into an application to open the window at an appropriate time, or evaluated in a workspace to launch the application.

Opening a Secondary Window

When the same application model serves one or more secondary canvases in addition to the main canvas, you can open a secondary canvas. A "secondary" canvas implies that the application has reached the proper state, that the instance variables required by the interface have been initialized.

This example creates a new UIBuilder the first time it is invoked, and it stores that builder in an instance variable. When your application needs to access widgets on the secondary canvas later, storing this second builder assures you will have a means of accessing the widgets.

1. In a method in the application model, create a new UIBuilder.
2. Tell the builder which object will supply its menus, aspects, and other resources by sending it a source: message. The argument is typically the application model itself. (Alternatively, you can send a series of aspectAt:put: messages to install the resources directly.)
3. Create the spec object and add the spec to the builder.
4. Open the window.

```
openFinder
    "Open the Search window. If already open, raise to top."

    | bldr |
    (self finderBuilder notNil and: [self finderBuilder window isOpen])
        ifTrue: [self finderBuilder window raise]
        ifFalse: [
            self finderBuilder: (bldr := UIBuilder new).
            bldr source: self.
            bldr add: (self class
                interfaceSpecFor: #finderSpec).
            bldr window
                application: self;
                beSlave.
            self adjustSearchScope.
            self searchStatus value: 0.
            (bldr componentAt: #searchStatus) widget
                setMarkerLength: 5.

            bldr openAt: (self
                originFor: bldr window
                nextTo: #findButton)].

    (self wrapperAt: #listView) takeKeyboardFocus.
```

Setting the Window Size at Opening

Rather than specify the opening size of a window in the canvas, you can specify the size when opening.

1. Build an interface up to the point of opening the window, by sending the allButOpenInterface: to an instance of the application model.
2. Get the window from the interface builder.
3. Ask the window to open with a specified size (extent), by sending the window an openWithExtent: message.

```
| bldr win |
bldr := Editor2Example new allButOpenInterface: #windowSpec.
win := bldr window.
win openWithExtent: 500@220.
```

Setting the Startup Location of a Window

1. Build the interface up to the point of opening the window.
2. Get the window from the interface builder.
3. Ask the window to open itself within a specified rectangle, using screen coordinates, in pixels.

```
| bldr win |
bldr := Editor2Example new
      allButOpenInterface: #windowSpec.
win := bldr window.
win openIn: (50@50 extent: win minimumSize).
```

Closing Application Windows

The platform window manager provides the user with a means of closing a window, such as clicking on a close icon. Often it is desirable for an application to provide a close method in its interface, such as an Exit option on a menu or a Quit button. An application might also close a window as a side effect of some conclusive user action, such as clicking the window manager's close icon.

The techniques shown here notify the window's model, so it can take precautions such as confirming the action.

When an application model is running one or more windows, you can close it (or all of them at once, if there is more than one) by sending closeRequest to the application.

Ask the application model to close its associated windows.

```
| editor |
editor := Editor2Example new.
editor openInterface: #windowSpec.
(Delay forSeconds: 1) wait.
editor closeRequest.
```

Hiding a Window

A window is a relatively expensive object, because it holds a visual component that is often bulky and because it allocates a display surface using the window manager. When your application needs to open and close a window repeatedly, it is not necessary to reconstruct it each time. Instead, you can unmap it, which hides the window without disassembling it. Then you can simply map it to redisplay it.

```
| win |
win := (Editor2Example open) window.
win display.
(Delay forSeconds: 1) wait.
win unmap.
(Delay forSeconds: 1) wait.
win map.
```

All windows also respond to an isMapped message, and keep their mapped state as a Boolean in their mapped instance variable.

Performing Final Actions

When an application window has been asked to close, it first sends a changeRequest message to its application model. If the model answers false, the window won't close; if it answers true, the window proceeds to close itself. Thus, the model has a chance to verify that no damage will be done if the window is closed.

For example, as shown below, the Image Editor (UIMaskEditor) uses a changeRequest method to confirm the user's intent to abandon any unsaved changes in the image.

Implement a changeRequest method in your application model, which answers true when the window can close and false otherwise.

```
changeRequest
    ^super changeRequest
        ifFalse: [false]
        ifTrue: [(self modified or: [self magnifiedBitView controller
  updateRequest not])
            ifTrue:
                [Dialog confirm: 'The image has been altered, but not installed.
                    Do you wish to discard the changes?']
            ifFalse: [true]]
```

Notice also in the example that the inherited version of changeRequest is first invoked to preserve any precautions that a parent class may have implemented.

3

Controlling the GUI Programmatically

An application frequently needs to exert control over the appearance and behavior of its user interface during runtime. This requires exerting programmatic control over the GUI. There are also times when control needs to be exerted at specific times during application startup and shutdown. Controlling the GUI requires accessing both the application window and the widgets it contains.

This chapter covers these issues.

Application Startup and Shutdown

The process of assembling and opening the application window proceeds by stages. The application model (ApplicationModel subclass):

1. Creates an instance of UIBuilder;
2. Sends itself preBuildWith: with the builder as argument;
3. Passes the UI specs to the builder and ask it to construct the UI objects;
4. Sends itself postBuildWith:, with the builder as argument;
5. Opens the fully assembled interface window;
6. Sends itself postOpenWith:, with the builder as argument.

The preBuildWith:, postBuildWith: and postOpenWith: messages each provide a way for the application model to intervene in the process to perform additional configuration.

Launching an Application

As described in the previous chapter, you typically open an application by sending an open or openWithSpec: message to the application model class. This creates an instance of the application model and opens the main window. To open using the default windowSpec specification, send an open message:

```
MyApp open
```

To specify an alternate specification, send an openWith: message:

```
MyApp openWithSpec: #myWindowSpec
```

The application model class creates a new instance of itself to run the interface.

As an alternative to defining a postBuildWith: message (described below), you can launch the application by sending the model class an allButOpenInterface: with the window specification as argument. This creates the builder which then builds the window, but does not open it. You can then make any adjustments to the UI as needed, and then open the window by sending finallyOpen to the application model.

If you want to use an existing application model instance, you can send open or openInterface: to that instance. This is useful when you want to reuse an instance rather than create a new one, or when you want to initialize the application specially.

Prebuild Intervention

After an instance of UIBuilder has been created, but before it has been given a window spec and built the UI, the application model sends itself a preBuildWith: message. The argument is the newly created UIBuilder.

Few applications need to intervene at this stage. Those that do, typically take the opportunity to load the builder with custom bindings that can only be determined at runtime, or must be set before building the UI. For example, if you need to assign a specific UILookPolicy, do so in this method:

```
preBuildWith: aBuilder
    aBuilder policy: MotifLookPolicy new
```

There are a few other examples in this document. Browse UIBuilder for appropriate messages.

Postbuild Intervention

Next, the UIBuilder creates a window according to the window specification. Before the builder opens the window, however, the application model sends itself a postBuildWith: message, with the builder as argument. The application model can use the builder to access the window and any widgets within the window that were given an **ID** property.

Applications commonly use postBuildWith: to hide or disable widgets, menu items, and toolbar items as needed by the runtime conditions:

```
postBuildWith: aBuilder
    "Disable the trip meter, making it read-only."
    (self wrapperAt: #tripRange) disable.
```

It is also the common place to register event interests for widgets:

```
postBuildWith: aBuilder
    self widget: #nextButton when: #clicked send: #nextRandom to: self.
    self widget: #resetButton when: #clicked send: #resetSequence to: self
```

Several other examples are provided in this document.

Postopen Intervention

Finally, the builder opens the fully-assembled interface. At this stage, the application model is sent a postOpenWith: message, again with the builder as argument. As with postBuildWith:, the application can use the builder to access the window and its widgets. This time, however, those objects have been mapped to the screen, which makes a difference for some kinds of configuration.

For example, the FileBrowser model that drives the File List interface uses postOpenWith: to insert the default path in the window's title bar—something it could not do until after the window had been opened.

Application Cleanup

An application model often needs to take certain actions when the application is closed. For example, a word-processing application might need to ask the user whether edits that have been made to the currently displayed text should be saved or discarded.

Another common cleanup action is to break circular dependencies that would otherwise prevent the application from being garbage collected. For example, if application A holds application B, and vice versa, for the purpose of interapplication communications, neither would be removed from memory even after both of their windows were closed.

If the application user exits from the application by using a menu or other widget in the interface, the application model performs the exit procedure and can insert any required safeguards. But if the user exits by closing the main window, a special mechanism is needed to notify the application model.

The application model is held by the application window. When the window is about to be closed, its controller asks for permission from the application model, by sending a requestForWindowClose. The application model can redefine this method to perform any cleanup actions and then return true to grant permission or false to prevent the window from closing.

Additional cleanup can be performed using the finalization mechanism described in Chapter 13, "Weak Reference and Finalization," in the *Application Developer's Guide*.

Windows

An application GUI consists of a window and the widgets it contains, as well as all the code to make the GUI interact with the application model. The standard GUI is presented as one or more windows containing individual widgets, such as input fields, buttons, text displays, and so on. Accordingly, a window is a container for its widgets.

A window is a display surface on which a set of widgets display their contents.

Creating a Window

In VisualWorks, windows are typically created using the UIPainter, as described in Chapter 2, "Building an Application's GUI." The WindowSpec produced by the painter is used by a UIBuilder to build the window.

Windows come in three types:

- Normal windows, having full decorations (type #normal)
- Dialog windows, having a border but (depending on the window manager) typically fewer border widgets (type #dialog)
- Pop-up windows, having no decorations (type #popUp)

GUIs built as subclasses of ApplicationModel are typically normal windows, and subclasses of SimpleDialog of dialog windows. Pop-up windows are built manually.

A window can be created and opened programmatically by sending a variant of the openNewIn: message to the window class. The argument is a rectangle. For example, to create a popup window, use the openNewIn:withType: message:

```
ApplicationWindow openNewIn: (20@20 extent: 100@100)
    withType: #popUp
```

Class Hierarchy

The crucial window class hierarchy is:

```
Object
    GraphicsMedium
        DisplaySurface
            Window
                ScheduledWindow
                    ApplicationWindow
```

The window class used for an application is typically an instance of ApplicationWindow, and is the default for GUIs build by the UIBuilder from a window spec. The parent class, ScheduledWindow, is used in applications that predate the VisualWorks canvas-painting tools. ScheduledWindow provides much of the state and behavior upon which ApplicationWindow relies.

Still farther back in the ancestor chain is Window, which is now an abstract class, and lacks a controller to enable a user to control the window, and it does not respond to events. However, it does provide important foundational methods for ApplicationWindow.

An ApplicationWindow can be created without a component, but the resulting window does not reopen when opening a saved image. If you need to have this window stay between image transitions, simply give it a single component:

```
myWindow component: ComposingComposite new
```

Window Components

Windows are complex objects, with many constituent objects held in instance variables that specify the look and behavior of the window. Several of these are worth specially noting, in order to understand how to control the windows. The following objects are for ApplicationWindows, and are necessary for understanding the basic VisualWorks windowing architecture. Other objects are described in later sections.

Controller

An application window's controller, which is usually an instance of ApplicationStandardSystemController, provides event forwarding for closing the window to its application. The close and closeNoTerminate messages handle this notification.

Component

A window also has a component, which can be a single visual component such as a ComposedText but is usually a CompositePart that holds a hierarchy of widgets.

Event Sensor

A window has an event sensor for providing information to widget controllers about mouse activity, and a keyboardProcessor for providing information about keyboard activity.

Manager

A window has a windowManager, an instance of WindowManager, which holds an event queue. The WindowManger manages itself and the window or windows under its control.

The WindowManager class replaces the job the ControlManager class used to do (pre-7.1). Each window used to have its own event queue, and the single ControlManager instance, the global ScheduledControllers, centrally managed then all by telling each window when to process its next event.

Window Processes

Prior to 7.1, each window stored a queue of events sent to it, and processing of events was directed by a single instance of ControlManager, named ScheduledControllers. Accordingly, there was only a single UI process, the one run by ScheduledControllers. To allow for multiple UI processes, this mechanism was changed in 7.1.

Each window now has a WindowManager that holds an event queue, representing a single UI process. Each WindowManager can manage the events for one or more windows. Usually, only closely related windows, such as windows in master/slave relation, or windows and dialogs they raise, share a manager. Note that a dialog blocks only the those windows sharing its window manager.

For example, if you open a System Browser from the Transcript, both windows will have their own managers. However, if you then open, for example, a senders browser from the System Browser, the two browsers will share a manager. To verify this, press **<Control>-Y** to open the process

monitor and explore the processes by debugging processes before and after opening the new window, and look for the windows in each process. Processes with more than one window are sharing a WindowManager.

Whether a window is created with a new WindowManager or using an existing WindowManager is determined by the windowing environment's WindowManagerUsagePolicy. The policy can be set to either

- MakeNewWindowManagerUsagePolicy, or
- UseParentWindowManagerUsagePolicy.

The default policy is UseParentWindowManagerUsagePolicy.

To set the policy for a window, which will govern the manager for all windows subsequently spawned by that window, include the following in its postOpenWith: method:

```
postOpenWith: aBuilder
    ...
    self mainWindow windowManager activeControllerProcess
        environmentAt: #WindowManagerUsagePolicy
        put: MakeNewWindowManagerUsagePolicy new.
    ...
```

You can also set the policy for an individual new window, by setting the policy, opening the window, and then resetting the policy.

Yielding to Other Processes

When a window manager processes an event, it yields to all other managers, giving them an opportunity to process events. In some circumstances, an UI process can be active and unyielding, not processing an event, and so prevent events in other processes from being handled. For example,

```
10 timesRepeat: [ ScheduledControllers restore ]
```

when run in one process prevents all other UI processes from processing their events until the last iteration is completed. In this situation, you may need to specifically provide an opportunity for other processes to process.

Inserting Processor yield into the block will ensure that at least one event for each WindowManager will be processed:

```
10 timesRepeat: [
    Processor yield.
    ScheduledControllers restore ]
```

To ensure that all events in the queues for all WindowManagers are processed, you can send

```
<someWindow> windowManager processOutstandingEvents.
```

in the block, where <someWindow> is a known window.

If you do not have a window to which you can send the above message, you can send:

```
Processor activeProcess windowManager ifNotNil:
    [:value | value processOutstandingEvents].
```

This does not ensure that the WindowManagers will all process their events, because it is not certain that the active process is a UI process, in which no UI events are processed. Nonetheless, this can be useful.

Finally, you can force this all to happen in a specific UI process by enclosing the whole thing in a block and sending a uiEventFor: message:

```
[ ... ] uiEventFor: <aWindow>
```

You can also prevent other UI processes from processing any events prior to executing your code, by enclosing your code in a block and sending a uiEventNowFor: message:

```
[ ... ] uiEventNowFor: <aWindow>
```

Refer to "Adding an Event to the UI Event Queue" for more on these two messages and maintaining an event queue.

Accessing Window Components

When an ApplicationModel or subclass instance opens an interface specification, it creates an interface builder, which in turn creates the specified window and its contents. To access windows and their components, the usual approach has been through this builder object. The application would send a self builder message, together with a specific message to access the desired component, for example:

```
self builder window
```

or

```
self builder componentAt: aWidgetID
```

This protocol has been replaced by a new, preferred protocol that allows you to access these objects without first getting the builder. See "Accessor Methods" below for the protocol. Specifically, these should now be written:

```
self mainWindow
```

and

```
self wrapperAt: aWidgetID
```

The new protocol is preferred because in future releases the builder will be removed, to be replaced by a new GUI building framework. The old access approach will then be no longer valid.

While the implementation of the of this new protocol still accesses the builder, future implementations will not. The new protocol will be preserved in future releases, though the implementation will change.

Accessor Methods

The following methods allow easy lookup of widgets and widget components without having to go through the application's builder object. We suggest using these messgage instead of the self builder messages commonly used in VisualWorks applications.

wrapperAt: *aSymbol*
Answer the value of the named component at *aSymbol*. Typically gets a SpecWrapper or nil. In the case of a toolbar, it gets the actual ToolBar instance. This method is the ApplicationModel direct replacement for messages of the form:

```
self builder componentAt: aSymbol.
```

controllerAt: *aSymbol*
Answers the controller for the component associated with *aSymbol*. The answer may be nil or a Controller. In the case of a toolbar, it will be nil. This method is the ApplicationModel direct replacement for messages of the form:

```
(self builder componentAt: aSymbol) controller.
```

widgetAt: *aSymbol*
Answer the widget associated with *aSymbol*. Typically answers a kind of VisualPart, which may be nil. This method is the ApplicationModel direct replacement for messages of the form:

```
(self builder componentAt: aSymbol) widget
```

mainWindow
Answer the main window associated with this ApplicationModel instances. Typically answers a ApplicationWindow. May be nil if the window is not created yet. This method is the ApplicationModel direct replacement for messages of the form:

```
self builder window
```

windowMenuBar

Answers the instance of MenuBar associated with the main window. May be nil if the window is not mapped and opened, or if there is no menu bar associated with the main window.

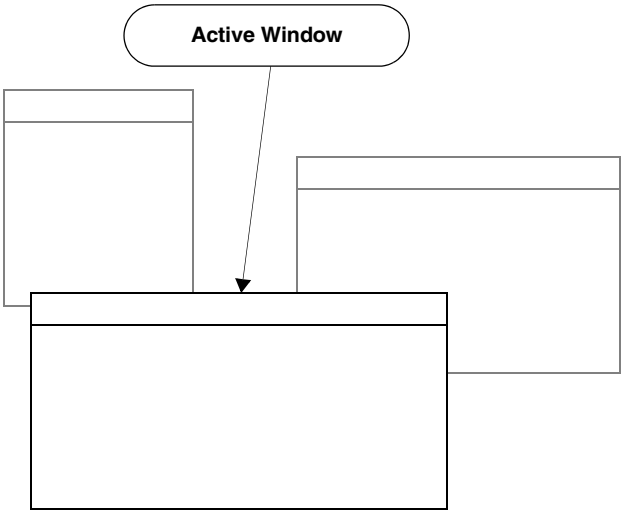
Accessing a Window

Getting an Application Window

Your application code can manipulate the window programmatically by obtaining the window from the application model and then sending it messages. To get the main window, send a mainWindow message to the application instance. For example, to get an application's window and change its label, do:

```
| app win |
app := Editor2Example new.
app open.
win := app mainWindow.
win label: 'Editor'.
```

Getting the Active Window

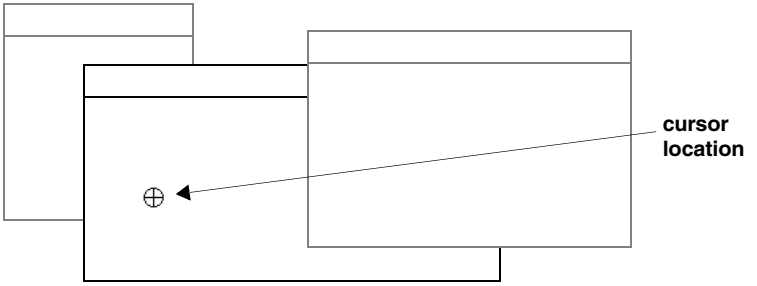


Class Window keeps track of the currently active VisualWorks window in the CurrentWindow shared variable. Its controller is the active controller. You can ask the active controller for its associated window.

To access a window, ask the active controller for its associated window, which is the topComponent associated with the controller's view.

```
| win |
win := Window activeController view topComponent.
win moveTo: 20@20.
```

Getting the Window at a Location



When your application performs an operation on a window that is pointed to by the user (using the mouse), you can access the window as shown here. Drag-and-drop operations, in particular, rely on this technique.

To get a window by its location:

1. Get the cursor location in screen coordinates by sending a globalCursorPoint message to the window controller's sensor.
2. Get the window at the cursor point by sending a windowAt: message to the default Screen. The argument is the cursor location. (In the example, the window's component flashes so you can verify that the correct window was accessed.)

```
| sensor pt window |
sensor := Window activeController sensor.
Cursor bull showWhile: [sensor waitButton].
pt := sensor globalCursorPoint.

window := Screen default windowAt: pt.

window component flash.
```

Closing a Window

An application will frequently have a button to close the application. To do so, simply get the window and send it a close message. For example, the might invoke this method:

```
exit
    self  mainWindow close
```

If there are more windows involved, you will have to ensure that they are all closed.

Setting Window Properties

Once you have a window, you can set a variety of properties for that window, overriding any properties set for the window when you created it. This provides for a great deal of programmatic control over windows.

The following sections describe how to set several properties. For other property setting options, browse the Window class and its subclasses.

Changing the Window Size

You can set a window size by giving it a new display box, using screen coordinates. You can also constrain the window size, as shown here.

```
| win |
win := (Editor2Example new) open; mainWindow.
win displayBox: (100@100 extent: 400@220).
```

Determining a Window's Dimensions

Several messages are available for determining the dimensions and constraints on a window. This example shows a few. For more, browse the Window class and its subclasses.

```
| win min max box origin width height |
win := (Editor2Example new) open; mainWindow.

min := win minimumSize.
max := win maximumSize.
box := win displayBox.
origin := box origin.
width := box width.
height := box height.
```

Changing a Window's Label

You can modify an open window's label by sending a label: message to the window, with the new label as argument.

```
| win |
win := (Editor2Example new) open; mainWindow.
win label: 'Editor'.
```

Adding and Removing Scroll Bars

Programmatically adding and removing scrollbars requires that scrollbars be initially activated, which is set in the Properties Tool for the window. Then, to remove scroll bars, send a noVerticalScrollBar or noHorizontalScrollBar message to the window's BorderDecorator. To add scrollbars, send a useVerticalScrollBar or useHorizontalScrollBar message:

```
| win |
win := ApplicationWindow new.
win component: (BorderDecorator
      on: Object comment asComposedText).
win open.

win component
      noVerticalScrollBar;
      noHorizontalScrollBar.
win display.

Cursor wait showWhile: [
      (Delay forSeconds: 2) wait].

win component
      useVerticalScrollBar;
      useHorizontalScrollBar.
```

Controlling Window Displays

A variety of operations can be performed on windows by their application, such as refreshing or collapsing (minimizing). For additional control operations, browse Window and its subclasses.

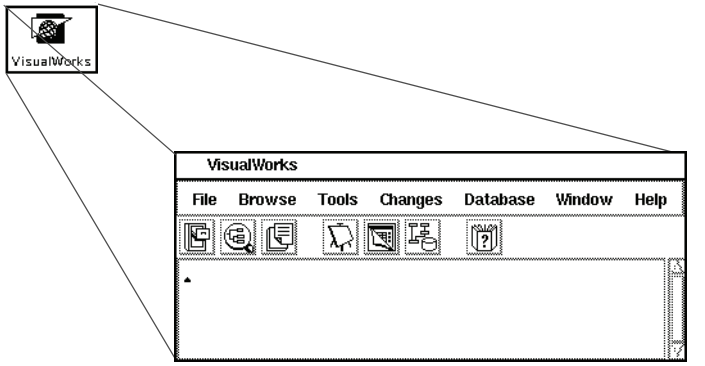
Refreshing a Window's Display

Under normal conditions, a window redisplays its contents whenever those contents change or whenever an overlapping window is moved. Sometimes you need to redisplay a window programmatically, as when you want to display an intermediate state of the window before a drawing operation has been completed.

To refresh a window, send it a display message:

```
| win |
win := (Editor2Example new) open; mainWindow.
5 timesRepeat: [
      (Delay forMilliseconds: 400) wait.
      win display].
```

Expanding and Collapsing a Window



Window managers typically provide a means of collapsing (iconifying) a window and expanding it back to its normal state. You can also control that behavior programmatically.

To collapse a window, send a collapse message to the window. To expand the window, send an expand message to the window.

```
| win |
win := (Editor2Example new) open; mainWindow.
win display.
(Delay forSeconds: 1) wait.
win collapse.
(Delay forSeconds: 1) wait.
win expand.
```

Assigning a Window Icon

For window managers that support iconified windows, VisualWorks provides a default icon to represent a collapsed window. You can assign a different icon to better represent the window. The icon must be an image (refer to “Graphical Images” in the Application Developer’s Guide).

Creating an Icon

An icon is an instance of Icon, and consists of an image (figure) and a mask (shape), both of which are instances of CachedImage. The icon may also be registered in the Icon class IconConstants dictionary.

To create an icon with a mask, send a figure:shape: message to an instance of Icon, specifying the image and mask:

```
| myIcon |
myIcon := Icon new
    figure: VisualLauncher CGHelp24
    shape: VisualLauncher CGHelp24Mask.
myIcon cleanFigure.
```

(The mask does not exist in the example, but can be created easily using the Image Editor.)

If either the image or mask is an instance of Image, not CachedImage, send asCachedImage to the image when assigning it to figure or shape.

The cleanFigure message is needed for color icons, to clean up some odd behavior on Windows systems, and may not be required in all cases.

Registering an Icon

Icons that are reused frequently can be registered with the Icon class, which stores icons in a dictionary.

To register an icon, send constantNamed:put: to the Icon class, with a symbol for its name and the icon as arguments:

```
| myIcon |
myIcon := Icon new
    figure: VisualLauncher CGHelp24
    shape: VisualLauncher CGHelp24Mask.
myIcon cleanFigure.
Icon constantNamed: #HelpIcon put: myIcon
```

To later retrieve the icon, send constantNamed: to the Icon class:

```
| win |
win := ScheduledWindow new.
win icon: ( Icon constantNamed: #HelpIcon).
win open
```

Installing an Icon

Instances of ScheduledWindow and its subclasses store an icon in their icon instance variable. To set an icon for a window, send an icon: message to the window with the icon to use:

```
| win |
win := ScheduledWindow new.
win icon: ( Icon constantNamed: #HelpIcon).
win open
```

Windows other than instances of ScheduledWindow or its subclasses do not hold onto the icon, and must ensure that the icon persists as long as the window does. Registering the icon is sufficient.

Slave and Master Windows

In a multiwindow application, it is often helpful to close all secondary windows automatically when the user closes the main window. In this situation, the main window is called the *master window* and the secondary windows are called *slave windows*.

Defining Slave and Master Windows

1. Tell the master window which application model to inform of its events.
2. Tell the master window to be a master.
3. Tell the slave window which application model will relay events from the master window.
4. Tell the slave window to be a slave.

```
| masterWin slaveWin |
masterWin := ( Editor1Example new ) openInterface; mainWindow.
masterWin
        label: 'Master';
        application: app;
        beMaster.

slaveWin := (Editor2Example new) open; mainWindow.
slaveWin
        label: 'Slave';
        application: app;
        beSlave.
```

Make Windows Equal Partners

When you want to be able to close all of your application's windows by closing any one of them, make them partners instead of master and slaves.

Tell the windows to be partners.

```
| win1 win2|

win1 := ( Editor1Example new ) openInterface; mainWindow.
win1
      label: 'Partner 1';
      application: app;
      bePartner.

win2 := (Editor2Example new) open; mainWindow.
win2
      label: 'Partner 2';
      application: app;
      bePartner.
```

Choosing the Events That Are Sent

By default, master and partner windows broadcast the following events: #close, #collapse, and #expand. You can remove any of those events, and you can add any of the following: #bounds, #enter, #exit, #hibernate, #reopen, and #release.

Tell the master or partner window which events to broadcast.

```
| masterWin slaveWin |
masterWin := ( Editor1Example new ) openInterface; mainWindow.
masterWin
      label: 'Master';
      application: app;
      beMaster;
      sendWindowEvents: #( #close #collapse
          #expand #hibernate #reopen).

slaveWin := (Editor2Example open) window.
slaveWin
      label: 'Slave';
      application: app;
      beSlave.
```

Choosing the Events That Are Received

By default, slave and partner windows mimic the following events: #close, #collapse, and #expand. Controlling the events that are received lets each slave be selective according to its needs.

Tell the slave or partner window which events to receive.

```
| masterWin slaveWin |

masterWin := ( Editor1Example new ) openInterface; mainWindow.
masterWin
    label: 'Master';
    application: app;
    beMaster.

slaveWin := (Editor2Example new ) open; mainWindow.
slaveWin
    label: 'Slave';
    application: app;
    beSlave;
    receiveWindowEvents: #( #close).
```

Window Events

Unlike widgets, windows and dialogs trigger many events that are directly related to platform operating system window events. Because of this, not all events are triggered on all platforms. Therefore, you must test your windows on all target platforms to ensure that the events you are registering occur when and how you expect.

Windows and dialogs also trigger events that originate within VisualWorks, rather than coming in from the operating system. These events are available to all Windows, without regard to the underlying operating system.

Platform based events are marked with the parenthetical (Platform) in their description. VisualWorks based events are marked with (VisualWorks) in their description.

An ApplicationModel can access the window or dialog by sending the #mainWindow message to itself. Then, the application can register an interest in the events below by sending when:send:to: messages.

#activate
(Platform) When the underlying platform sends an activate notification event to a VisualWorks window, the window triggers the #activate event. On some platforms, only the enter event is sent.

#bounds
(Platform) When the underlying platform changes the size of the VisualWorks window and sends the bounds notification event, the window triggers the #bounds event.

#clicked

(VisualWorks) Previously, a window did not know when it was clicked on. With this new event, any window which is clicked on with the <Select> mouse button, in any area where there is not another widget that handles the event, will cause the window to trigger the #click event. This event is not triggered in the Menubar or Toolbar areas. Labels and ViewHolders are examples of widgets that do not handle mouse clicking, and for these, the underlying window will trigger the #clicked event.

#close

(Platform) When the underlying platform sends a close notification event to a VisualWorks window, the window triggers the #close event.

#closing

(VisualWorks) When a window is in the process of closing, but before the window is closed and loses focus, the window sends the #closing event.

#collapse

(Platform) When the underlying platform sends a collapse notification event to a VisualWorks window, the window triggers the #collapse event.

#deactivate

(Platform) When the underlying platform sends a deactivate notification event to a VisualWorks window, the window triggers the #deactivate event. On some platforms, only the exit event is sent.

#destroy

(Platform) When the underlying platform sends a destroy notification event to a VisualWorks window, the window triggers the #destroy event.

#doubleClicked

(VisualWorks) As with the #clicked event, any window which is double clicked with the <Operate> mouse button on any area where there is not another widget that handles the event, will cause the window to trigger the #doubleClicked event. This event is not triggered in the Menubar or Toolbar areas. Labels and ViewHolders are examples of widgets that do not handle mouse clicking, and for these, the underlying window will trigger the #doubleClicked event.

#enter

(Platform) When the underlying platform sends an enter notification event to a VisualWorks window, the window triggers the #enter event.

#exit

(Platform) When the underlying platform sends an exit notification event to a VisualWorks window, the window triggers the #exit event.

#expand

(Platform) When the underlying platform sends an expand notification event to a VisualWorks window, the window triggers the #expand event.

#expose

(Platform) When the underlying platform sends an expose notification event to a VisualWorks window, the window triggers the #expose event.

#gettingFocus

(VisualWorks) This event is triggered whenever a window that does not have focus, is given top visual focus, either by clicking in the window, clicking on one of the windows visual accessories (such as the title bar or a border), or by using an operating system feature to bring the window to top focus.

#losingFocus

(VisualWorks) This event is triggered whenever a window that has top focus is asked to no longer be the main window focus. This may happen by clicking in or on any another window (without regard to if it is a VisualWorks window), or by using an operating system feature to bring another window to top focus. In the case of VisualWorks Dialog windows, the Dialog window will not loose focus to VisualWorks windows until it is closed, and therefore will not trigger the #losingFocus event until that VisualWorks Dialog is closed.

#mapped

(VisualWorks) This is the second VisualWorks event that all window trigger. It occurs just after the initial display surface is visually brought to life. This also is triggered when a window which has been programmatically unmapped, is remapped.

#menuBarCreated

(VisualWorks) At the time an ApplicationModel receives the #postBuildWith: message, no menu bar has been created. This makes it hard to register to a menu bar's events in the postBuildWith: method. The developer has two choices, either register to a menu bar's events in the postOpenWith: message, or register with the window's #menuBarCreated event and do menu bar event registration at that time. Once a menu bar is created, it can be access by the new windowMenuBar method of the ApplicationModel.

#middleClicked

(VisualWorks) As with the #clicked event, any window which is clicked with the <Window> mouse button on any area where there is not another widget that handles the event, will cause the window to trigger the #middleClicked event. This event is not triggered in the Menubar or Toolbar areas. Labels and ViewHolders are examples of widgets that do not handle mouse clicking, and for these, the underlying window will trigger the #middleClicked event. Windows and Dialogs are the only visual objects in VisualWorks that trigger this event.

#mouseEnter

(VisualWorks) This event is triggered when the mouse is moved, but not dragged, into the bounds of a window.

#mouseExit

(VisualWorks) This event is triggered when the mouse is moved, but not dragged, out of the bounds of a window.

#move

(Platform) After a window is moved, the underlying platform sends a move notification event to a VisualWorks window, the window in response triggers the #move event.

#opening

(VisualWorks) This is the very first VisualWorks event that all window trigger. It occurs before the window is mapped and before it gets focus.

#resize

(Platform) When the underlying platform sends a resize notification event to a VisualWorks window, the window triggers the #resize event.

#rightClicked

(VisualWorks) As with the #clicked event, any window which is clicked with the <Operate> mouse button on any area where there is not another widget that handles the event, will cause the window to trigger the #rightClicked event. This event is not triggered in the Menubar or Toolbar areas. Labels and ViewHolders are examples of widgets that do not handle mouse clicking, and for these, the underlying window will trigger the #clicked event.

#scrollDown

(VisualWorks) If a window has vertical scroll bars, and the window view is made to scroll down, the window triggers the #scrollDown event.

#scrollLeft

(VisualWorks) If a window has horizontal scroll bars, and the window view is made to scroll to the left, the window triggers the #scrollLeft event.

#scrollRight

(VisualWorks) If a window has horizontal scroll bars, and the window view is made to scroll to the right, the window triggers the #scrollRight event.

#scrollUp

(VisualWorks) If a window has vertical scroll bars, and the window view is made to scroll up, the window triggers the #scrollUp event.

#toolBarCreated

(VisualWorks) As with the menu bar, at the time an ApplicationModel receives the postBuildWith: message, no tool bar has been created. This makes it hard to register to a tool bar's events in the postBuildWith: method. The developer has two choices, either register to a menu bar's events in the postOpenWith: message, or register with the window's #menuBarCreated event and do menu bar event registration at that time. Once the tool bar is created, it can be accessed by the widgetAt: method using the ID of the tool bar as defined in the window properties page of the UIPainter tool.

#unknownEvent

(Platform) There are many raw events that come into a VisualWorks window from the underlying operating system. Any operating system event that arrives at a VisualWorks window that VisualWorks does not otherwise handle, is always sent to the main window as an UnknownEvent. There are many ways of trapping these otherwise unhandled events in the GUI framework. This event is triggerd as a notification of when one of these events arrives. It is important to note that the format of the underlying raw operating system event is different for each platform, and that it is thus up to the developer to deal with any cross platform issues. Additionally, these events come in fast and furious.

#unmapped

(VisualWorks) This event is triggered whenever a window is programmatically unmapped. This event does not occur during the sequence of closing a window.

Registering Window Events

As for events in general, an application registers an interest in a window event by sending a variant of the when:send:do: message. The interest is usually registered in the application's postOpenWith: message, and sends the message to its main window. For example, to register an interest in the #mouseEnter event, include a line like the following in the application model postOpenWith: method:

```
self mainWindow when: #mouseEnter send: #notify to: self.
```

Adding an Event to the UI Event Queue

DeferrableAction is an event class that allows one process to add an event to the event queue of a UI process. This mechanism is provided to allow a non-UI process to add an event to a UI process, and to allow communication between UI processes. DeferrableAction encapsulates a message send that can be put on a window manager's event queue to be executed.

DeferrableAction has a rich API which allows you to create complex actions that can be executed in the UI process. The class side has two instance creation methods, which are similar to the message signatures used in the Trigger Event system:

send: *selectorSymbol* **to:** *anObject*
Creates a DeferrableAction to send a *selectorSymbol* unary message to *anObject* when invoked.

send: *selectorSymbol* **to:** *anObject* **with:** *aCollection*
Creates a DeferrableAction to send a *selectorSymbol* keyword message to *anObject* when invoked, with *aCollection* of parameters for the message.

An instance of DeferrableAction allows you to specify the window (using window:, which is inherited from Event) for which you want to have the action taken. If a window is not specified when the action is invoked, it makes a best guess attempt to find the current window. Since this guess can be mistaken, it is preferred to specify the window.

To invoke an instance of DeferrableAction send one of these messages:

activate
Puts the action on the queue of the window without waiting for a response.

waitForResult
Puts the action on the queue of the window, and waits for the result of the message.

Four class side methods allow you to create, specify the window, and invoke the action, all in a single message:

send: *selectorSymbol* **to:** *anObject* **for:** *aWindow*

send: *selectorSymbol* **to:** *anObject* **with:** *aCollection* **for:** *aWindow*

sendNow: *selectorSymbol* **to:** *anObject* **for:** *aWindow*

sendNow: *selectorSymbol* **to:** *anObject* **with:** *aCollection* **for:** *aWindow*

The meaning of these messages is clear from the preceding discussion.

Four methods are provided to allow you to put a BlockClosure in a UI event queue.

uiEvent
: Puts the block in the event queue of the current window, at which it makes a best guess, without waiting for a response.

uiEventFor: *aWindow*
: Puts the block in the event queue of aWindow, without waiting for a response.

uiEventNow
: Puts the block in the event queue of the current window, at which it makes a best guess, then waits for a response, but without blocking the window.

uiEventNowFor: *aWindow*
: Puts the block in the event queue of aWindow, then waits for a response, but without blocking the window.

uiEvent and uiEventFor: put the receiver block into the event queue and do not wait for an answer. uiEventNow and uiEventNowFor: put the receiver block into the event queue, and then, without blocking the window, separately wait for the result of the block being evaluated.

Because uiEvent and uiEventNow can only take a best guess at which is the current window, uiEventFor: and uiEventNowFor: are the preferred messages.

Controlling Widgets

It is often useful for an application to be able to access the widgets in its windows. Depending on the purpose of the access, a widget may be accessed directly or through its wrapper. In either case, the widget is identified by its ID, which is set in the Properties Tool in the Canvas.

Accessing a Widget

In a System Browser, edit a method in the application model (in this example, alignCenter) so that it sends a widgetAt: message to the application model. The argument is the **ID**.

```
alignCenter
    | widget style |
    widget := self widgetAt: #comment .
    style := widget textStyle copy.
    style alignment: 2.
    widget textStyle: style.
    widget invalidate.
```

Accessing the Widget's Wrapper

Online example: HideExample

In some cases, the application model must send messages to the *wrapper* that surrounds the widget. A wrapper is an instance of WidgetWrapper, which controls various aspects of the widget's appearance, such as visibility, enablement, and layout.

1. In a canvas, select the widget to be accessed. In the widget's **ID** property, enter an identifying name for the widget. Apply the properties and install the canvas.
2. In a System Browser, edit a method in the application model (in this example, changedListVisibility) so that it sends a wrapperAt: message to the application model. The argument is the **ID**.

```
changedListVisibility
    | wrapper desiredState |
    wrapper := self wrapperAt: #colorList.
    desiredState := self listVisibility value.

    desiredState == #hidden
        ifTrue: [wrapper beInvisible].

    desiredState == #disabled
        ifTrue: [
            wrapper beVisible.
            wrapper disable].
    desiredState == #normal
        ifTrue: [wrapper enable; beVisible].
```

Setting Widget Properties

The properties that are typically set in the Canvas using the Properties Tool can also be set or changed during an application run by sending appropriate messages to the widget or its wrapper. The following sections explain how to set a number of properties. Additional options are available, and can be found by browsing the widget classes.

Changing a Widget's Size

In some circumstances, your application may need to resize a widget while the application is running. In Size3Example, a colored region is resized in response to **Expand** and **Shrink** buttons.

Online example: Size3Example

1. Get the widget's wrapper from the application model.
2. Send a bounds message to the wrapper to get the widget's existing size.
3. Create a rectangle having the desired origin and extent, using the widget's bounding rectangle to derive the new values.
4. Send a newBounds: message to the wrapper. The argument is the new bounding rectangle.

```
expandBox
    | wrapper oldSize newSize |
    wrapper := self wrapperAt: #box.
    oldSize := wrapper bounds.

    "If the box is bigger than the window already, do nothing."
    oldSize origin x < 0
        ifTrue: [^nil].

    "Expand the bounding rectangle by 10 pixels on each side."
    newSize := Rectangle
        origin: oldSize origin - 10
        corner: oldSize corner + 10.

    "Assign the new bounding rectangle to the widget wrapper."
    wrapper newBounds: newSize.
```

Changing a Widget's Font

Online example: Font1Example

1. In a method in the application model, get the widget from the application model.

2 Create an instance of TextAttributes corresponding to the new font. If the font exists in the fonts menu, you can send a styleNamed: message to the TextAttributes class. The argument is the name of the font (for example, #large for the system's Large font).

3 Get the label from the widget by sending a label message; get the text of the label by sending a text message to it. Then install a blank text temporarily as a means of erasing the old label if the new font is smaller.

4 Install the new font in the widget by sending a textStyle: message to the widget. The argument is the TextAttributes you created in step 2.

5 Reinstate the original label by sending a labelString: message to the widget.

```
changedFont
    | widget newStyle oldLabel |
    widget := (self widgetAt: #label) .
    newStyle := TextAttributes styleNamed: (self labelFont value).

    "Erase the existing label in case its font is larger than the new one."
    oldLabel := widget label text.
    widget labelString: ''.

    "Install the new font."
    widget textStyle: newStyle.

    "Reinstate the original label."
    widget labelString: oldLabel.
```

Hiding a Widget

Sometimes a widget is useful only under certain conditions and needs to be hidden at other times to avoid confusing the user of your application. Action buttons need to be hidden when their actions are not appropriate.

A widget may also be hidden when two alternative widgets are layered on top of each other. For example, the Online Documentation window uses a text editor on top of a list editor and hides the view that is unneeded at any given time.

You can turn on a widget's **Initially Invisible** property to cause the widget to be hidden when the window opens. You can also program the application model to hide and show the widget while the application is running.

Online example: HideExample

1 In a method in the application model, get the widget's wrapper from the application model.

2 To hide the widget, send a beInvisible message to the wrapper.

3 To make the widget visible again, send a beVisible message to the wrapper.

```
changedListVisibility
    | wrapper desiredState |
    wrapper := self wrapperAt: #colorList.
    desiredState := self listVisibility value.

    desiredState == #hidden
        ifTrue: [wrapper beInvisible].

    desiredState == #disabled
        ifTrue: [
            wrapper beVisible.
            wrapper disable].

    desiredState == #normal
        ifTrue: [
            wrapper enable.
            wrapper beVisible].
```

Disabling a Widget

Sometimes a widget is useful only under certain conditions, but making it invisible would be confusing to the user of your application. You can disable a widget, causing it to be displayed in gray. In addition, its controller is inactivated so the widget does not respond to user input. Action buttons are frequently “grayed out” when not needed.

You can turn on a widget’s **Initially Disabled** property to cause the widget to be disabled when the window opens. You can also program the application model to disable and enable the widget while the application is running.

Online example: HideExample

1 In a method in the application model, get the widget’s wrapper.

2 To disable the widget, send a disable message to the wrapper.

3 To make the widget active again, send an enable message to the wrapper.

```
changedListVisibility
    | wrapper desiredState |
    wrapper := self wrapperAt: #colorList.
    desiredState := self listVisibility value.

    desiredState == #hidden
        ifTrue: [wrapper beInvisible].

    desiredState == #disabled
        ifTrue: [
            wrapper beVisible.
            wrapper disable].

    desiredState == #normal
        ifTrue: [
            wrapper enable.
            wrapper beVisible].
```

Changing a Widget's Colors

Online example: ColorExample

1. In a method in the application model, get the widget's wrapper.
2. Get the LookPreferences from the wrapper and create a copy with the desired color. The copy is created when a color-zone message is sent: foregroundColor:, backgroundColor:, selectionForegroundColor:, or selectionBackgroundColor:. The argument is the desired new color.
3. Install the new LookPreferences by sending a lookPreferences: message to the wrapper. The argument is the new LookPreferences.

```
foregroundColor: aColor
    "For each sample widget, change the indicated color layer."

    | wrapper lookPref |
    self sampleWidgets do: [ :widgetID |
        wrapper := (self wrapperAt: widgetID).
        lookPref := wrapper
            lookPreferences foregroundColor: aColor.
        wrapper lookPreferences: lookPref].
```

Adding and Removing Dependencies

When a widget's value is changed, such as when an item is selected from a list, the application often needs to react in some way. A common reaction is to update other widgets based on the new value. You can arrange for such a reaction, typically as part of the initialization process. This is known as *setting up a dependency* or *registering an interest*.

You can also bypass the dependency when unusual circumstances arise. For example, when two widgets depend on each other, one of them must bypass the dependency mechanism to avoid infinite recursion.

Adding a Dependency

Online example: DependencyExample

In the application model's initialize method (typically), send an onChangeSend:to: message to the widget's value holder. The first argument is a message, which will be sent to the second argument. The second argument is typically the application model itself.

```
initialize
    colorNames := SelectionInList with: ColorValue constantNames.
    selectedColor := String new asValue.
    fieldIsDependent := false asValue.

    "Arrange for the application model to take action when the
    check box is turned on or off."
    fieldIsDependent
        onChangeSend: #changedDependency to: self.
```

Removing a Dependency by Retracting the Interest

Online example: DependencyExample

1. Send a retractInterestsFor: message to the widget's value holder. The argument is the object that registered the interest, typically the application model itself.
2. After the value has been changed, register the interest again.

```
changedDependency
    "Turn on or off the dependency link between the list and
    the input field, depending on the value of the check box."

    | valueModel |
    valueModel := self colorNames selectionIndexHolder.

    self fieldIsDependent value
        ifTrue:
            [valueModel onChangeSend: #changedSelection to: self]
        ifFalse:
            [valueModel retractInterestsFor: self].
```

Bypassing All Dependencies

Online example: FieldConnectionExample

1 Send a setValue: message to the widget's value holder instead of the usual value: message. The argument is the widget's new value.

2 Get the widget from the application model and ask the widget to update itself with the new value.

```
changedB
    "Use setValue: to bypass dependents, thus avoiding circularity."
    self bSquared setValue: (self b value raisedTo: 2).

    "Since dependents were bypassed when the model was updated,
    update the view manually."
    (self widgetAt: #b2) update: #value.
```

Validation Properties

Frequently, only certain entries are valid for a particular widget. For example, you might want to restrict input accepted by an input field to a numeric range from 0 to 999, or check for incompatible checkbox selections.

Validation properties specify messages a widget sends to its application model asking for permission to proceed. In this way, validation provides input flow control, for example, to prevent the user from entering invalid data into an input field, or to prevent the user from entering a field before filling in other prerequisite fields.

Property	Description
Entry	Specifies the message the widget sends to its application model when it prepares to accept focus. If the method returns true, the widget takes focus; otherwise, focus is refused.
Change	Specifies the method the widget sends to its application model after the user changes the widget's value and attempts to exit the widget before the widget writes the input value to its value model. The method should determine whether the input value is acceptable. If the method returns true, the widget's controller writes the input value to the value model; otherwise, the value model remains unchanged.
Exit	Specifies the method the widget sends to its application model when it prepares to give up focus. The message is sent any time the user attempts to exit the widget. The method should determine whether the widget can actually give up focus. If the method returns true, the widget gives up focus; otherwise, focus is retained.
D. Click	Specifies the method the widget sends to its application model when preparing to respond to a double-click.

Validation methods, which you add to the application model, return either true or false. On true, the widget proceeds with its action; on false, the widget waits for new, valid input. You can implement the validation method to redirect input focus or to disable and enable input widgets.

To create a validation method that inspects the widget's value, specify a selector with a colon (selector:). The widget passes its controller object as the argument to the method. The method can then ask the controller for the widget's value. For certain widgets (input fields and combo boxes), you use statements such as the following to get and set values through the controller:

```
input := aController editValue
```

Notification Properties

You specify Notification properties when you want a widget to inform its application model that certain actions have taken place, namely, that the widget has taken focus, changed internal state, or given up focus. Notification properties are useful for facilitating complex flow of user input.

Each notification property specifies the symbolic name of a notification callback, which is the message you want the widget to immediately send after the relevant action. For each notification callback you specify, you must program the application model to contain a corresponding method.

You implement this method to provide the desired response to the widget's action. You can implement the notification method to activate other widgets in the interface.

Property	Description
Entry	Specifies the symbolic name for the widget's entry notification callback. The widget sends this message to its application model immediately after taking focus. You must implement a corresponding method in the application model that provides the desired response to this event.
Change	Specifies the symbolic name for the widget's change notification callback. The widget sends this message to its application model immediately after the widget sends its input value to its value model. You must implement a corresponding method in the application model to provide the desired response to this event. Note that specifying a change notification callback is similar to registering an interest in a value model via `onChangeSend:to:`, in that both cause a message to be sent after the value in the value model has changed. However, the two techniques also differ in important ways: The change notification callback is sent only to the application model. The message specified by `onChangeSend:to:` is sent to the specified receiver, which may, but need not be the application model. If both techniques are used together, the message sent by `onChangeSend:to:` is sent first, and the change notification callback is sent second.
Exit	Specifies the symbolic name for the widget's exit notification callback. The widget sends this message to its application model immediately after it gives up focus. You must implement a corresponding method in the application model to provide the desired response to this event.
D. Click	Specifies the symbolic name for the widget's double-click notification callback. This callback is the message that the widget sends to its application model in response to a double-click. You must implement a corresponding method in the application model to provide the desired response to this event. This property appears with the List and Table widgets only.

Giving a Widget Keyboard Focus

It is occasionally desirable to control which widget has keyboard focus following an action. For example, when an action button is clicked, by default it then has keyboard focus. But, it may be more reasonable for the focus to return, say to a list or a text editor, without having to click back in that widget.

To set keyboard focus, send a message such as the following:

```
self mainWindow keyboardProcessor requestActivationFor:
    (self controllerAt: #List1).
```

This asks the keyboard processor to activate (give focus to) the controller for, in this case, the list widget, #List1.

4

Adapting Domain Models to Widgets

Introduction

As described in Chapter 2, “Building an Application’s GUI”, for widgets that have an aspect, the value of the aspect is represented by its value model. Value models provide a powerful mechanism for establishing dependencies between objects, so that when a change occurs to one object, another is automatically notified and updated.

While this mechanism is useful between any objects, it is used most to establish dependencies between domain models and application models. The value model mechanism is a fundamental feature of the VisualWorks application framework, and is essential to the design and building of VisualWorks applications.

This chapter describes some standard approaches to configuring value models.

Value Models

A widget that presents data (such as an input field) relies on an auxilliary object, called a *value model*, to manage the data it presents. That is, instead of holding onto the data directly, a data widget delegates this task to its value model. Thus, when a data widget accepts input from a user, it stores this data in its value model. When a data widget needs to update its display, it asks its value model for the data to be displayed.

The VisualWorks value model mechanism provides a uniform set of messages for accessing the data to be presented, allowing all data widgets to store and refresh their data in a standard way. Two messages are central to the value model:

value
Returns the data value from the value model.

value: *anObject*
Sets data value in the value model and sends a changed: message with the new value.

Other objects, such as the application model, can also send these messages to a value model to obtain or change a widget's data programmatically.

A data widget is a *dependent* of its value model, in the sense that the widget depends on its value model to notify it when the relevant data has changed. The widget responds to such notification by asking the value model for the new data and displaying it. This keeps the widget's display synchronized with changes made programmatically to the data.

Choosing a Value Model

There are three standard value model objects used in VisualWorks, implemented by these three classes:

- ValueHolder
- AspectAdaptor
- PluggableAdaptor

There are several other special-purpose adaptors as well. Browse the ValueModel subclasses for the full collection.

As described in Chapter 2, "Building an Application's GUI", the **Define** operation in the UI Painter creates a simple value model for each widget using a ValueHolder. This mechanism holds the aspect value in an instance variable of the application model class. The **Define** operation adds an instance variable to the class and creates a stub accessor method for the variable, which returns the ValueHolder. The ValueHolder responds appropriately to the value and value: messages.

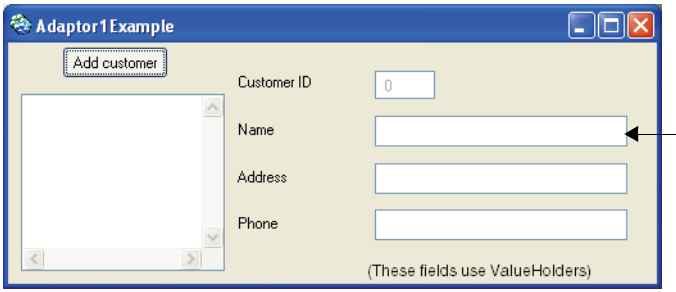
You can alway coordinate values in a domain model and those held in an aspect instance variable, and so use a ValueHolder. In early development, this gives a way to set up a GUI and verify that it operates properly, possibly using only test values. In an application with a domain model well separated from the application model, using a ValueHolder requires that you provide the logic necessary to coordinate values between the models.

For many applications, using a ValueHolder is not the preferable approach, because it involves maintaining a value twice: in both the domain and application models. Rather, it is frequently better to configure the application model to get at least some of its widgets' values directly from the domain model. This is done by using an AspectAdaptor, which redirects the value and value: messages to the domain model. In addition to reducing the overhead of the application model, by eliminating instance variables, getting values from the domain model is also frequently simpler than ensuring that the domain and aspect values are coordinated.

When the model does not provide the protocol needed to use an AspectAdaptor, a PluggableAdaptor allows you to specify accessor blocks to provide additional processing. This provides the convenience of working with an adaptor while specifying more of the GUI-specific processing in the application model.

The trigger-event system provides an additional approach to defining dependencies, and is appropriate in some situations, particularly for ensuring that domain and application models are coordinated. Most windows, dialogs, and widgets trigger events in specified conditions. Configuring dependencies using events is described later in this chapter, under "Configuring Dependencies Using Events". For a general description of the trigger-event system, refer to the *Application Developer's Guide*, chapter 6, "Event System." The events triggered by each widget are described in Chapter 9, "Configuring Widgets."

Configuring a ValueHolder



A ValueHolder is the most basic type of value model. As its name implies, a ValueHolder is a holder of an object, which is its value. It's primary purpose is to respond to the value and value: messages, as expected by a widget. As such, it is the appropriate value model for data that is held in an instance variable within the application model itself.

Online example: Adaptor1Example

Adaptor1Example maintains a list of customer information, which is represented as instances of Customer1Example. The list itself is held by the application model, and each record is represented by instance variables in the application model. So, this example is suitable for using ValueHolder as its value models.

To initialize a variable as storing a ValueHolder, send an asValue message to the object that is to be held (in the example, accountID is initialized to a ValueHolder holding the number 0).

```
initializeID
    accountID := 0 asValue.
    accountID onChangeSend: #changedID to: self.
```

The onChangeSend:to: message invokes additional processing whenever the value is changed by sending a value: message to the ValueHolder. It is not required by the ValueHolder, but is used in this example to update the display.

If the value is a String, you can send a newString message to the ValueHolder class, which is equivalent to the expression String new asValue. The choice of which to use is a matter of personal preference.

```
initializeName
    name := ValueHolder newString.
    name onChangeSend: #changedName to: self.
```

Similarly,

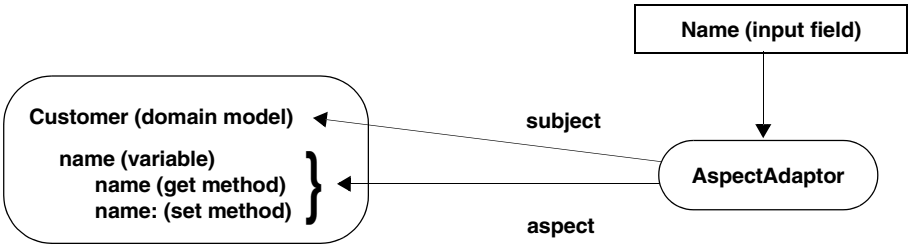
- to initialize the value to the Boolean, false, you can send a newBoolean message to the ValueHolder class. This is equivalent to the expression false asValue.
- to inititalize the value to the Fraction, 0.0, you can send a newFraction message to the ValueHolder class. This is equivalent to the expression 0.0 asValue.

Once a variable has been set to a ValueHolder, update its value by sending a value: message, as in this segment from the changedCustomer method:

changedCustomer

```
...
        self accountID value: chosenCustomer accountID.
        self name value: chosenCustomer name.
        self address value: chosenCustomer address.
        self phoneNumber value: chosenCustomer phoneNumber.
...
```

Configuring an AspectAdaptor



When a widget presents data that is modeled in a domain object outside of the application model, it is not necessary to hold that data in a variable in the application model. While you can represent such data in a variable in the application model, and so use a ValueHolder as described above, it is often better to get the data directly from the domain object. In this case, it is appropriate to use an AspectAdaptor as the value model, instead of a ValueHolder. An AspectAdaptor is conceptually a pointer to the remote data.

An AspectAdaptor has a *subject*, which is the target domain model, an *accessor selector*, and an *assigner selector*, which are sent to the subject when, respectively, the value and value: messages are sent to the adaptor.

To initialize an AspectAdaptor, there are several protocols available, but a simple approach is:

```
| aspect |
    aspect := AspectAdaptor subject: aDomainObject.
    aspect accessWith: #getSelector assignWith: #putSelector
```

Because Smalltalk conventions recommend that an instance variable instVar have get and put accessor methods instVar and instVar:, this can be simplified in cases where the convention is followed to:

```
| aspect |
    aspect := AspectAdaptor subject: aDomainObject.
    aspect forAspect: #iVarName
```

Frequently, the subject needs to be changed. To do this, send a subject: message to the adaptor with the new subject object as argument:

```
aspect subject: newDomainObject
```

Several aspects may use the same subject, for instance if several parts of the same complex object need to be accessed.

Configuring an AspectAdaptor with a Subject

For example, Adaptor1Example can be modified so that the individual customer fields are not held locally, but retrieved directly from the Customer1Example objects using their own protocol. To make the change, you would:

1 Change each aspect variable initialization method to initialize to an AspectAdaptor, rather than the ValueHolder. For example:

```
initializeID

    accountID := AspectAdaptor subject: customers selection.
    accountID forAspect: #accountID.
```

Note that the aspect accessor methods will now return an AspectAdaptor.

2 Edit the changedCustomer message to update the subject for each adaptor, and remove unneeded expressions:

```
changedCustomer

    | chosenCustomer selector |
    chosenCustomer := self customers selection.
    self accountID subject: chosenCustomer.
    self name subject: chosenCustomer.
    self address subject: chosenCustomer.
    self phoneNumber subject: chosenCustomer.

    chosenCustomer isNil
        ifTrue: [
            self accountID value: 0.
            self name value: ''.
            self address value: ''.
            self phoneNumber value: '']
        ifFalse: [
            self accountID value: chosenCustomer accountID.
            self name value: chosenCustomer name.
            self address value: chosenCustomer address.
            self phoneNumber value: chosenCustomer phoneNumber].

    #( #accountID #name #address #phoneNumber)
        do: [ :componentName |
            (self builder componentAt: componentName)
                isEnabled: chosenCustomer notNil].
```

This is sufficient for a simple change to the example, though it lacks two features: displaying an accountID of 0 when no customer is selected, and updating the list display as soon as the accountID or name fields are edited. These defects are easily repaired, and are left as an exercise for the reader.

Configuring an AspectAdaptor with a Subject Channel

Online example: Adaptor2Example

When several AspectAdaptors share the same subject, and especially if the subject changes frequently, it is simpler and less error prone to assign each adaptor a *subject channel* rather than simply a subject. This is the case in the modification of Adaptor1Example outlined above, and so a subject channel is used in Adaptor2Example and the subsequent adaptor examples.

A subject channel provides indirect access to the subject, using a ValueHolder. Any time the value of the subject channel is changed, it is changed for all adaptors sharing it, thus simplifying the process of updating the subject.

To initialize an AspectAdaptor with a subject channel, send a subjectChannel: instance creation method, with a ValueHolder containing the channel. Accessor and assigner methods are identified as before. For example:

```
| aspect |
    aspect := AspectAdaptor subjectChannel: aValueHolder.
    aspect accessWith: #getSelector assignWith: #putSelector
```

Again, if conventional accessor and assigner names are used, this can be simplified to:

```
| aspect |
    aspect := AspectAdaptor subjectChannel: aValueHolder.
    aspect forAspect: #iVarName
```

Typically, the channel ValueHolder will be held in an initialized instance variable. Then, whenever its value is updated, the channel is updated for all applicable adaptors.

Configuring a subject channel is illustrated in the Adaptor2Example application, as follows:

1 In an initialize method in the application model, initialize an instance variable (selectedCustomer) with a ValueHolder on the domain model (a Customer1Example instance).

```
initialize

    customers := SelectionInList new.
    customers selectionIndexHolder
        onChangeSend: #changedCustomer to: self.
    selectedCustomer := Customer1Example new asValue.
```

2 In each aspect accessor method (for example, accountID), send a subjectChannel: message to the AspectAdaptor class, with the ValueHolder created in step 1 as argument. Also, assign the aspect accessor selectors.

```
accountID

    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
    adaptor forAspect: #accountID.
    adaptor onChangeSend: #redisplayList to: self.
    ^adaptor
```

3 In any method that would change the subject of the adaptors, update the subject channel by sending a value: message with the new subject:

```
changedCustomer

    | chosenCustomer |
    chosenCustomer := self customers selection.

    self selectedCustomer value:
        (chosenCustomer isNil
            ifTrue: [Customer1Example new]
            ifFalse: [chosenCustomer]).

    "Enable/disable selection-sensitive widgets."
    #( #accountID #name #address #phoneNumber #format)
        do: [ :componentName |
            (self builder componentAt: componentName)
                isEnabled: chosenCustomer notNil].
```

In this example, the subject channel is updated in two cases. The first assigns a default Customer1Example instance when there is no currently selected customer simply to provide appropriate display values. The second assigns the currently selected customer, so its current values are displayed and will be updated if modified. In both cases, the subject is updated for all aspect adaptors in the application that have that subject channel.

Adapting Unconventional Accessors

Online example: Adaptor2Example

By Smalltalk convention, accessor message selectors for getting and setting the value of an instance variable can be derived from the variable's name. For example, the domain model in Customer1Example provides name and name: methods for accessing the value of its name variable. The previous examples relied on this when identifying the accessor methods by sending a forAspect: message to an adaptor:

```
accountID

    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
    adaptor forAspect: #accountID.
    adaptor onChangeSend: #redisplayList to: self.
    ^adaptor
```

However, this is only a convention, and many domain models do not follow it, but provide accessor methods whose names are different from the instance variable they access. For example, another common naming pattern is, for an instance variable called income, to name its accessors getIncome and putIncome:.

To specify these accessor method selectors, send an accessWith:assignWith: message to the adaptor, instead of forAspect:. The first argument is the selector of "get" accessor method, and the second argument is the selector of the "set" assigner method.

Adaptor2Example does not require this form, since the selectors follow the usual convention, but illustrates how to use this form in configuring the AspectAdaptor for the address aspect:

```
address
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
    ^adaptor
        accessWith: #address
        assignWith: #address:
```

Adapting a Changing Domain

Online example: RandomWatcher and RunawayRandoms

An aspect adaptor notices programmatic changes to the data upon receiving a value: message. However, it is common for a domain model to change its data independently of the application model, and so without sending any messages to the AspectAdaptor. In this case, you need to provide a way for the domain model to notify the application model when it changes a value on which the application model depends.

This involves two modifications: the application model must be alerted to changes initiated by the domain, and the domain model must send out a notification when the change occurs.

The example classes illustrate how a domain model, RunawayRandoms, which updates the value of its current instance variable independently of the application model, RandomWatcher. RunawayRandoms was first developed without signalling its change. RandomWatcher was then created, but could not initially be updated. RunawayRandoms was then modified, while running, by editing the next method to send changed:, at which point RandomWatcher immediately started showing the updates.

1. Send a subjectSendsUpdates: message to the adaptor with the argument true. This causes the adaptor to register itself as a dependent of the subject. In RandomWatcher:

```
currentValue
    | adaptor |
    adaptor := AspectAdaptor subject: generator.
    adaptor forAspect: #current.
    adaptor subjectSendsUpdates: true.
    ^adaptor
```

2. In the domain model class (RunawayRandom), edit every method that alters the data value directly to send a changed: message to self. This notifies all dependents when the domain model makes the relevant change.

```
next
    current := generator next.
    self changed: #current
```

Configuring a PluggableAdaptor

An AspectAdaptor requires accessor methods in the domain model that correspond to an aspect in the application model. Occasionally there is no such direct correspondence, so additional processing is required to adapt the application to the domain. While you can do this processing in the application model and hold the results in a ValueHolder, using a PluggableAdaptor provides the capability but with the advantages of an AspectAdaptor. Even when there is such a correspondence, a PluggableAdaptor can be used to provide additional processing of the aspect values, if needed.

A PluggableAdaptor takes three blocks, which enable it to perform custom actions at three junctures in the flow of communications between the widget and the domain model:

- The getBlock: controls what happens when a value is fetched from the model by the widget. This is a one-argument block with the model as argument.
- The putBlock: controls what happens when a value is sent to the model by the widget. This is a two-argument block with the model and value as arguments.
- The updateBlock: controls when the widget updates itself based on an update message sent by the model. This is a three-argument block with the model, aspect being updated, and the update parameter as arguments. The block returns false if the update is to be rejected, or true if the update is to be accepted.

When the widget is the only source of changes to the data value, the update block can simply return false. When the data value can be changed by other objects, the update block performs a test to determine whether the widget should refetch the data value. Typically this test uses the update block's second argument, the aspect, to find out whether the aspect that it cares about has changed and, if so, accepts the update.

Create a PluggableAdaptor by sending an on: message to the class, with the model as argument:

```
PluggableAdaptor on: aModel
```

The model is the object from which the adaptor receives its values, so may be either the application or domain model. In the case of a domain model, it is equivalent to the AspectAdaptor subject.

As with an AspectAdaptor, the model may change. You can change the adaptor model by sending a model: message to the PluggableAdaptor instance:

```
adaptor model: newModel
```

And, if the model is shared by several adaptors, you can assign it a subject channel, so all adaptor models are changed together:

```
adaptor subjectChannel: aValueHolder
```

Configuring Accessor Blocks

Online example: Adaptor6Example

In this example, a PluggableAdaptor is used to translate an integer such as 342 into a string containing prefixed zeroes ('000342'), saving the user the trouble of entering the leading zeroes.

1. Create the custom adaptor by sending an on: message to the PluggableAdaptor class. The argument is the domain model.
2. Send a getBlock:putBlock:updateBlock: message to the adaptor. The first block takes one argument: the domain model. The second block takes two arguments: the model and the value to be assigned. The third block takes three arguments: the model, the aspect of the model that was changed, and a parameter that corresponds to the argument of a changed:with: message, and returns a Boolean.

```
paddedID
    | paddedID |
    paddedID := PluggableAdaptor on: self.
    paddedID
        getBlock: [ :model |
            | paddedString |
            paddedString := model accountID printString.
            6 - paddedString size
                timesRepeat: [paddedString := '0', paddedString].
            paddedString]
        putBlock: [ :model :value |
            model accountID: value asNumber]
        updateBlock: [ :model :aspect :parameter | false].
    ^paddedID
```

Setting the update block to return false causes the adaptor to reject all updates from either the subject domain or application model. Only changes entered into the entry field are accepted.

Configuring the Update Block

The update block is invoked whenever an update is submitted to the adaptor from the domain or application model. The block must evaluate to a Boolean; if true, then the update is accepted, and if false, the update is rejected. Accordingly, the block provides an opportunity to test an update to determine whether or not to accept it.

When invoked, the block must handle three arguments, which are the model, the aspect, and any parameter passed with the update.

For illustration purposes, RandomWatcher can be modified to use a PluggableAdaptor instead of an AspectAdaptor. Since it needs to accept updates from its domain model, the update block must provide an appropriate response. The only method that needs to be changed is:

```
currentValue
    | adaptor |
    adaptor := PluggableAdaptor on: generator.
    adaptor
        getBlock: [ :model | model current ]
        putBlock: [ nil ]
        updateBlock: [ :model :aspect :parameter |
            (model = generator) & (aspect = #current) ].
    ^adaptor
```

The model argument of the update block is the object causing the update, in this case the domain model. If the update were caused by the application model, by sending a value: message to the aspect, the model would be self. In this way you can discriminate between updates by their source.

Similarly, the aspect argument is the aspect that has changed. In the RunawayRandoms next method, this is specified as #current in its changed: message. This allows discrimination between updates based on the aspect being changed.

Finally, the parameter is the value of the change. Using this argument, you can accept or reject, or invoke special processing, based on the value. For example, you could test for value ranges and invoke special processing for out-of-range data.

Synchronizing Updates (BufferedValueHolder)

Frequently, when a domain model contains several values that are updated from a set of widgets, it is desirable to delay updating the domain model until all of the widgets in the set are ready to update their models. This is especially true in applications that make use of a database, to ensure that an entire data record, or row, is updated at once synchronously. BufferedValueHolder provides this capability.

A BufferedValueHolder wraps a value model, providing a buffer between the value model and its widget. When a value is entered at the widget, that that value is held in the buffer until it is triggered to update the value model.

Each BufferedValueHolder holds a value model as its subject, and a *trigger channel*, which is a ValueHolder containing true or false. When set to true, the trigger channel notifies its dependent widgets to update their models. When set to false, the BufferedValueHolder discards the buffered values, canceling the update.

Note that the buffering occurs in only one direction: from the widget to the value model. If a domain model updates itself and notifies its dependents, as described in "Adapting a Changing Domain" above, the update to the widgets display is immediate.

Adding a BufferedValueHolder

Online example: Adaptor3Example

Adaptor3Example provides an **OK** button that the user presses after entering or editing customer information in the input fields. When pressed, the button action sets the trigger channel to true, signalling all of the BufferedValueHolders to update their subject value models. If a customer is deselected before the update is triggered, the values are cleared from the buffers.

Adaptor3Example is derived from Adaptor2Example by making these changes:

1 In the application model, create an instance variable (updateTrigger) to contain the true/false value, and add an accessing method for the variable:

```
updateTrigger
    ^updateTrigger
```

2 Initialize the variable to a ValueHolder containing false:

```
initialize
    customers := SelectionInList new.
    customers selectionIndexHolder
        onChangeSend: #changedCustomer to: self.
    selectedCustomer := Customer1Example new asValue.
    updateTrigger := false asValue.
```

This simply sets the initial value of the trigger channel.

3 For each widget in the series, wrap it's value model in a BufferedValueHolder by sending a subject:triggerChannel: message to the BufferedValueHolder class. The first argument is the widget's value model (in the example, an AspectAdaptor). The second argument is the trigger channel (updateTrigger).

```
accountID
    | adaptor bufferedAdaptor |
    adaptor := AspectAdaptor subjectChannel: self selectedCustomer.
    adaptor forAspect: #accountID.

    bufferedAdaptor := BufferedValueHolder
        subject: adaptor
        triggerChannel: self updateTrigger.

    ^bufferedAdaptor
```

Note that the buffered value holder does not replace the widget's value model, but wraps it.

4 Add an Action Button for the user to indicate that the set of values is complete and ready to be accepted. Install the canvas and provide the action method (accept), which sets the value true to the trigger channel (updateTrigger), thereby triggering the update

```
accept
    self updateTrigger value: true.

    self redisplayList.
```

Discarding the Buffered Values

Online example: Adaptor3Example

To clear the buffered values without updating the models, set the value of the trigger channel to false. Note that the action is caused by setting the value to false; that the value is already false has no effect.

In Adaptor3Example, the buffers are cleared without updating if the selected customer is changed or deselected from the list before clicking **OK**. This is done in the changedCustomer method:

```
changedCustomer

    | chosenCustomer |
    chosenCustomer := self customers selection.

    self selectedCustomer value:
        (chosenCustomer isNil
            ifTrue: [Customer1Example new]
            ifFalse: [chosenCustomer]).

    "Discard changes that were not OK'd."
    self updateTrigger value: false.

    "Enable/disable selection-sensitive widgets."
    #( #accountID #name #address #phoneNumber #ok)
        do: [ :componentName |
            (self builder componentAt: componentName)
                isEnabled: chosenCustomer notNil].
```

Typically in a UI, a confirmation dialog would be opened before clearing the buffers, allowing the user to cancel the action. The example does not do this.

A good UI design would also generally provide a way to explicitly clear the buffers, rather than leaving the action implicit as in the example. To make it explicit, you could add another button, and give it an action method setting the trigger channel to false, forcing the buffers to clear:

```
reject

    self updateTrigger value: false.
```

Adapting Collections

Collections of many types are commonly used in domain models to hold associated data items, and often need to be represented in a GUI. The appropriate value model to select for the collection depends on a variety of factors, including:

- the kind information the domain model class provides on the collection (does it hold a “selected” item?),

- the protocol provided by the domain model class to access that data (does it provide an accessor method for the item to be displayed in your GUI?), and
- the requirements of the specific widget used to display the data (e.g., a List widget needs a SelectionInList, and a Tree widgets needs a SelectionInTree).

If the domain model already has a protocol providing access to the data, then it is reasonable to use an AspectAdaptor. If it does not, then you have to choose whether to add protocol to the domain model, use a PluggableAdaptor, or use a ValueHolder and represent the needed information as needed within the application model. Or, you can use one of the special adaptors, or create your own.

Adapting to a SelectionInList

Online examples: Adaptor4Example and Customer2Example

A list or notebook widget is designed to work with a SelectionInList as its aspect, which contains value holders for its collection and current selection. A domain model that contains a collection can be adapted to provide the list, by assigning an adaptor to its list holder. Typically, the domain will not maintain a selection in that collection, which is only of interest to the application model.

Like the previous examples, Adaptor4Example uses a SelectionInList for the List widget aspect, but modifies the earlier examples by adapting the list held in a Customer2Example instance, which holds an OrderedCollection of customers.

In the application model, add an instance variable (customers) for the List widget's aspect, and one for the current selection (selectedCustomer). Initialize the List widget's aspect variable (customers) to a SelectionInList. Then make the SelectionInList list holder an AspectAdaptor on the domain model:

```
initialize
    collectionModel := Customer2Example new.

    customers := SelectionInList new.
    customers listHolder:
        ( (AspectAdaptor subject: collectionModel) forAspect: #customers).

    self customers selectionIndexHolder
        onChangeSend: #changedCustomer to: self.
    selectedCustomer := Customer1Example new asValue.
```

The selection is set to an empty Customer1Example simply to provide default display values.

Also supply the usual aspect accessor method that simply returns the variable's value:

```
customers
    ^customers
```

The initialization could also be done as lazy initialization in the accessor method.

In the special case where the domain class does hold the selection and provide accessor methods for the selection, you can set up the adaptor by sending an adapt:aspect:list:selection: message to the SelectionInList class. The adapt: argument is the domain model (in the example, collectionModel). The aspect: argument is typically the name of the domain model's collection variable. The list: argument is the name of the domain model's method that returns the collection. The selection: argument is the name of the domain model's method that sets the selection in the collection.

Adapting a Indexable Collection

When the domain model is an indexable collection or an object with indexed instance variables, the appropriate adaptor is an IndexedAdaptor. An IndexedAdaptor redirects the value message to at: and value: to at:put:, making it simple to adapt an indexed collection to the requirements of the application model.

An IndexedAdaptor has a subject or subject channel, and specifies an index number in place of the aspect accessors.

In Adaptor5Example, the domain model has a vector that holds an instance of Vector, an object with 3 indexed instance variables representing *x*, *y*, and *z* coordinates.

Online example: Adaptor5Example, Vector

To configure an IndexedAdaptor:

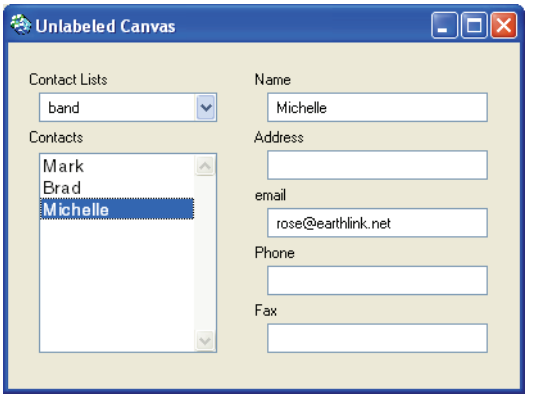
1. Send a subject (or subjectChannel:) message to the IndexedAdaptor class, with the indexed object (or a ValueHolder on the indexed object) as the argument.

2 Send a forIndex: message to the adaptor. The argument is the position of the desired element in the collection.

```
xAxis
    | adaptor |
    adaptor := IndexedAdaptor subjectChannel: self vector.
    ^adaptor forIndex: 1
```

Adapting Collections of Collections

Online example: Contact1Browser



Domains can become rather complex, and there are many options for adapting them. The main issue is one of adapting the right domain object to the appropriate part of the widget model.

The Contacts Book example provides a domain consisting of individual Contact instances, which are collected into one or more ContactsLists, and one or more contact lists collected into a ContactBook which is implemented as a Dictionary. The browser, Contact1Browser, needs to adapt each of these.

Note that the example is incomplete in some respects, but contains the essential adaptors.

First, the application holds its contact book in an instance variable, book, which is initialized in the initialize method. This is the highest level domain object and is the primary access point to the others.

In the application, we want to select a group to display, for which we use a ComboBox widget. The widget needs a list of choices and an aspect which is the current selected item in the list, named groupList and group, respectively. The aspect, group, is provided by the widget when a

selection is made. A change notification method, specified on the Notification property page, handles the update. The list of groups itself, however, must come from the book, which we can get using an adaptor.

The book holds contact lists, each with a group label, which is the key for the list in a Dictionary. The groups method in ContactBook returns the Set of those keys, and we can adapt this to provide the groupsList value. Since a Set is not appropriate, we need to modify the value returned, and so use a PluggableAdaptor:

```
groupsList

    | listAdaptor |
    listAdaptor := PluggableAdaptor on: book.
    ^listAdaptor
        getBlock: [ :model | model groups asOrderedCollection ]
        putBlock: [ :model :adaptor | nil]
        updateBlock: [ :model :adaptor :para | false ].
```

The application at this point does not update the book (no lists or groups can be added), so the put block does nothing and the update block returns false, because no updates will come from the domain.

When a group is selected from the list, the group aspect variable is assigned a value, and the list of contacts needs to be provided. The list of contacts is displayed in a List widget, the aspect for which, contactsList, is a SelectionInList. The list is provided by the ContactList corresponding to the selected group, where the group is the key to the list in the book. Accordingly, the list holder is provided as an adaptor on a ContactList.

The list will change frequently, whenever the selected group changes. To effect the change, the change notification for group (mentioned above), which we called changedGroup, changes the list whenever group changes:

```
changedGroup
    "When the selected group changes, change the list."

    contactsList list: (book listFor: group value) contacts asList.
    contactChannel value: Contact new.
```

Finally, when the specific contact is selected, an adaptor is needed on the Contact to extract the displayable information. This changes frequently, so a subject channel is used, and updated whenever the contact selection is changed. A change notification is set for the List widget to send changedContact when a new contact is selected:

```
changedContact
    "When a new contact is selected, update the information"

    contactChannel value: contactsList selection
```

The, for each contact aspect, an adaptor is defined on that subject channel. For example, for the name aspect, the adaptor is defined as:

```
name
    | adaptor |
    adaptor := AspectAdaptor subjectChannel: contactChannel.
    adaptor forAspect: #name.
    ^adaptor
```

This illustrates one way of setting up adaptors to access data from nested collections. Any of the adaptors could be replaced with value holders, at the cost of more processing in the application model. And, the adaptors would be configured differently if more or less protocol were provided by the domain model, possibly requiring more PluggableAdaptors. Extensions and modifications to the example are left as an exercise for the reader.

Defining Adaptors in the UI Painter

Instead of defining a value model in the application model, as either a ValueHolder or an adaptor, the UI Painter allows you to specify an *aspect path* in the **Aspect** property field.

The aspect path causes the interface builder to create an appropriate aspect adaptor to connect the widget to its domain model. The path may also cause the builder to create an input buffer behind the widgets, by combining the aspect adaptor with a BufferedValueHolder.

An aspect path has two parts: the *head* and the *path*. The head begins the aspect path, and is the name of the accessor method that returns the value model holding the domain object being adapted. The path is one or more aspect selectors, specifying how to access the value of the domain model. Each path element may be an accessor message selector or an index, depending on the object type.

In addition to the head and path, you can include a *trigger* that buffers the values for the aspects to update all buffered widgets at once.

Each of these is illustrated in the following sections.

Aspect Path with Aspect Selectors

Online example: Contact2Browser

Contact2Browser modifies Contact1Browser by replacing the aspect adaptors for each of the input field aspects with an aspect path. This simplifies the example by eliminating several aspect methods. Then, a few other modifications are needed to support the aspect paths.

1 Change the aspects of the input field specs to the following:

ID	Aspect
#name	#contactChannel name
#address	#contactChannel address
#email	#contactChannel email
#phone	#contactChannel phone
#fax	#contactChannel fax

In each of these, the aspect path head is contactChannel, which is a method selector we need to implement, and the path is the aspect selector: name, address, email, phone, and fax. Each of these identifies an aspect accessor in class Contact, an instance of which will be picked out by contactChannel.

2 Implement contactChannel to return the currently selected contact:

```
contactChannel
    ^contactsList selectionHolder
```

These are all the necessary changes, but there are several cleanup tasks we can perform:

- Remove the name, phone, fax, email, address, and changedContact methods.
- Remove the line that set contactChannel in changedGroup, to clear the contact information, which is no longer needed:

```
changedGroup
    "When the selected group changes, change the list."

    contactsList list: (book listFor: group value) contacts asList
```

- Remove changedContact method, and remove the reference to it from the List widget **Notification** property page.

- Change initialize, eliminating the initialization of contactChannel:

```
initialize
    book := ContactBook new.
    contactsList := SelectionInList new.
    group := String new asValue.
```

- Remove the contactChannel instance variable from the class definition.

Aspect Path with Index Selectors

Online example: Adaptor8Example

When the domain model is an indexable collection or an object with indexed instance variables, the aspect path can be the index rather than an accessor method. Adaptor8Example is a modification of Adaptor5Example that illustrates this.

In this example, the accessor method vector returns the domain model, and instance of Vector, as was the case Adaptor5Example. The only difference is in the aspect specification for the three input fields.

Coordinate	Aspect
x	#vector 1
y	#vector 2
z	#vector 3

The aspect path for each field identifies the index location of the object returned by vector, replacing the need for accessor methods for the coordiantes.

Aspect Path with Input Buffering

Online example: Adaptor7Example

When you need buffered input, to synchronize updates, you add a *trigger* following the path. The trigger is preceeded by a vertical bar, or pipe (|), followed by the triggering method selector.

Adaptor7Example modifies Adaptor3Example by using aspect paths with triggers instead of explicitly configured BufferedAdaptors.

In this example, the four input fields are configured with aspect paths with triggers, as follows:

ID	Aspect
#accountID	#selectedCustomer accountID | updateTrigger
#name	#selectedCustomer name | updateTrigger
#address	#selectedCustomer address | updateTrigger
#phoneNumber	#selectedCustomer phoneNumber | updateTrigger

The selectedCustomer method returns an instance of Customer1Example, which supports the accessor methods accountID, name, address, and phoneNumber. The accept method sets the value of updateTrigger to true, triggering the update, as in Adaptor3Example.

With the use of aspect paths, the accountID, name, address, and phoneNumber methods are not needed in the Adaptor7Example class, and so are not implemented.

The example incorporates some addition changes as well, involving using the notification property for changing the selected customer, but this change is independent of the use of aspect paths.

Configuring Dependencies Using Events

The Trigger-Event mechanism provides a much finer-grained approach to configuring interactions between the GUI and the application and domain models, by providing many more occasions for a response to be evoked.

For example, using the usually dependency mechanism, an ActionButton widget sends its action message when the button is clicked, but using events the widget can evoke a response when it is clicked, pressed, tabbed into or out of, getting or losing focus, or when its label is changing. Similarly, while an InputField widget updates its model when its value is changed, using events it can also evoke a response when its value is changed, just before the change is accepted, when it is clicked, right-clicked, or double-clicked, when it is scrolled, and so on.

Registering an Interest in a Widget Event

As described in chapter 6, “Event System,” in the *Application Developer’s Guide*, an interest in an event can be registered with any object at any time. In building the GUI, however, interests are typically registered as soon as the UI is built, and rarely need to be changed after that.

Accordingly, the most usual place to register event dependencies is in the application model postBuildWith: method, which is invoked by the UI Builder immediately after building the window.

For example, it is simple to modify the WalkThrough example, RandomNumberPicker, to use events rather than the original dependency mechanism. To illustrate with the two action buttons, remove the **Action** item for each in the GUI Painter Tool, assign each an **ID** (**#nextButton** and **#resetButton**), and re-install. Then add the following method to the application model:

```
postBuildWith: aBuilder
    self widget: #nextButton when: #clicked send: #nextRandom to: self.
    self widget: #resetButton when: #clicked send: #resetSequence to: self
```

Test the application to verify that it works as before.

Similar changes can be made to use events for updating the entry field widgets.

Update Notifications using Events

Using events with the dependency mechanism is a useful alternative to the subjectSendsUpdates: dependency mechanism, especially if your domain is already configured to trigger events on changes. Using events in this context requires a small change in the adaptor configuration.

Online Example: RandomWatcher and RunawayRandoms

In the section "Adapting a Changing Domain" above, RandomWatcher uses the dependency mechanism. To switch to the event system requires two changes. In the domain model, trigger an event instead of sending the notification:

```
next
    current := generator next.
    self triggerEvent: #currentChanged with: current.
```

The event symbol can be whatever you want to call the event. It triggers the event with the value of current.

Then, in the application model, register an interest in the event with the domain model:

```
currentValue
    | adaptor |
    adaptor := AspectAdaptor subject: generator.
    adaptor forAspect: #current.
    generator when: #currentChanged
        do: [ :arg | adaptor update: #current with: arg from: generator ].

    ^adaptor
```

Because the argument is important, register with a when:do: message. The single argument is passed into the do: block, and the adaptor value is updated. In the update message, the update: argument is the aspect, the with: argument is the new value, and the from: argument is the source of the value. If the from: argument is the subject for the aspect, the update is accepted.

The trigger-event mechanism can be used with PluggableAdaptor as well, as long as the update block returns true. The method would then be:

```
currentValue
    | adaptor |
    adaptor := PluggableAdaptor on: generator.
    adaptor
        getBlock: [ :model | model current ]
        putBlock: [ nil ]
        updateBlock: [ :model :aspect :parameter |
            (model = generator) & (aspect = #current) ].

    generator when: #currentChanged
        do: [ :arg | adaptor update: #current with: arg from: generator ].
    ^adaptor
```

Registering an interest in the event can be done in other methods. In this case, note that the update block arguments are provided by the registration message, so the aspect is #current as specified.

5

Menus

VisualWorks provides menu bars, menu buttons, and pop-up menus as GUI elements for presenting lists of options. All three elements use an underlying instance of Menu for their options. The Menu Editor provides an easy-to-use tool for creating a menu, or you can assemble a menu programmatically.

Menu bar and pop-up menus by default send a message to the application model, while a menu button by default places a value in its aspect value holder. Building a menu for these different behaviors affects what you assign to each menu item while building the menu.

To override the default behavior for a type of menu, you can define the value of a menu item in an action block. In this way, you can cause a menu bar or pop-up menu item to place a value in a value holder, or a menu button item to execute a command. This makes the VisualWorks menu support extremely flexible while at the same time providing a powerful framework for menu implementation.

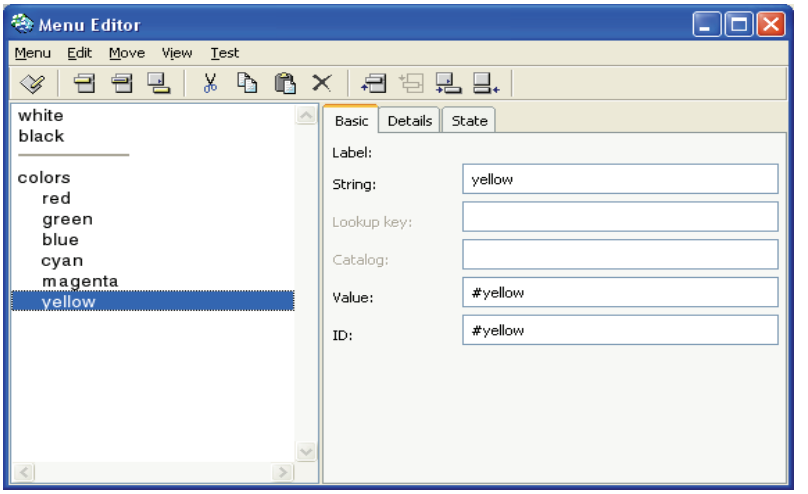
Creating a Menu

You can create a menu either using the Menu Editor, or programmatically.

With the Menu Editor, you can create menus of messages or values. If a more complex action is required, such as a compound message or an action block, you can assemble the menu programmatically.

Complex actions are necessary, for example, for assigning a value to a value holder from a menu bar or pop-up menu, or executing a command from a button menu.

Creating a Menu using the Menu Editor



The Menu Editor provides both menu selections and buttons for several design operations. In the following instructions, we refer to the menu selections, but the button equivalent can be used.

Online example: MenuEditorExample

1 Open a Menu Editor by selecting **Tools → Menu Editor** in the GUI Painter Tool or **Painter → Menu Editor** in the Launcher.

2 Add a top-level menu by selecting **Edit → New Item**, and edit its properties.

When you add a menu item, it is displayed as **<new item>**. Edit this label in the **Label String** property, giving the menu item the label you want displayed.

To include a mnemonic for the menu item, insert an ampersand (&) character in the label before the letter to use as the mnemonic. For example, **I&tem** makes *t* the mnemonic letter for the item, and is indicated in the menu by underlining the letter.

Subsequent uses of **Edit → New Item** add items at the same level as the selected item.

3 Add a submenu by selecting a label and choosing **Edit → New Submenu Item**.

The new item is placed below and indented from the selected item.

4 In each item's **Value** property, either:

- enter a message selector
- leave blank, if the item has submenu items

A message selector is sent to the application model by menu bar and pop-up menus. For a button menu, the selector should be the widget's aspect accessor method, which updates the aspect value holder.

A value property is not needed if the item has subitems, since selecting the item only displays the submenu.

5 Specify the shortcut character, if desired.

A shortcut character allows the user to select this menu item by pressing the character key while holding down one or more modifier keys (<Ctrl>, <Alt>, or <Shift>). To specify a character, enter it in the **Shortcut character:** field on the **Details** page, and check the desired modifier key check boxes. The shortcut will be displayed in the menu.

6 Adjust any menu item levels and locations as needed.

Move → Left and **Move → Right** change the menu level of an item. **Move → Up** and **Move → Down** change the order of items.

7 If the item needs to be enabled and disabled during application execution, enter an Enablement Selector on the **State** page. The method you write for the selector must return a Boolean value.

8 To use a menu item selection indicator, a check mark by the item, enter an Indication Selector on the **State** page. You can also select for each item whether its selector is initially on or off. The method you write for the selector must return a Boolean value.

9 Choose **Menu → Install…** to install a specification for the menu in a resource method of the application model.

10 Create any methods required by message names entered in a menu item's **Value** property.

Creating a Menu Programmatically

When building a menu programmatically, each menu label is associated with its action, which may be a value, a command, or an action block.

Since value and command menus can be created using the Menu Editor, we illustrate the programmatic approach with an action block example (but see the MenuCommandExample and MenuValueExample online examples).

Action blocks are typically used in menu bar and pop-up menus that place values in value holders (as illustrated), or in button menus that execute a command. To execute a command, you configure the value holder for the button menu to send perform: with its value holder value as argument when its value is changed.

A menu typically is created by a method in a resources protocol on the class side of an application model. The resource method can also be an instance method, which is necessary when it relies on data supplied by a running application.

Online example: MenuValueExample

1 In a menu-creating instance method, create a menu builder by sending a new message to the MenuBuilder class.

An instance method is used in the example (templatesMenuForMenuBar) because information is needed from the application model instance. If this is not required, the menu creation method can be a class method (see the MenuValueExample class resource methods).

2 For each menu item, send an add: message to the menu builder with an association as the argument.

The association relates the menu item's label string and the block that performs the required action. For value and command menus, the value holder or message selector occur, as symbols, in place of the block.

To include a mnemonic for the menu item, insert an ampersand (&) character in the label before the letter to use as the mnemonic. For example, I&tem makes *t* the mnemonic letter for the item, and is indicated in the menu by underlining the letter.

3 To add a shortcut key sequence for the menu item, which displays on the menu, specify the character key and the modifier key or keys. (Note that this is not illustrated in the example code.)

For example, to specify <Ctrl>+<Shift>+A as the shortcut, add these lines:

```
menuItem shortcutKeyCharacter: $A.
menuItem shortcutModifiers:
    (InputState ctrlMask bitOr: InputState shiftMask).
```

The modifier keys can be specified individually (ctrlMask, shiftMask, or altMask), or combined using bitOr: as shown above.

4 To insert a submenu, send a beginSubMenuLabeled: message to the menu builder with the submenu label as argument.

This begins a submenu definition. Add submenu items by sending an add: message to the menu builder, as before.

At the end of the submenu definition, send an endSubMenu message to the menu builder.

5 Get the menu from the menu builder using a menu message and return it as the result of the menu-creating method.

In this example, all menu items are added within a submenu. This is necessary here because the menu is being added to the menu bar. The submenu label is displayed in the menu bar.

The example also illustrates adding graphics and color programmatically.

```
templatesMenuForMenuBar
    | mb menu submenu |
    mb := MenuBuilder new.

    mb
        beginSubMenuLabeled: 'Templates';
        add: ' ' -> [self letter value: self class firstNotice];
        add: ' ' -> [self letter value: self class secondNotice];
        add: ' ' -> [self letter value: self class finalNotice];
        endSubMenu.

    "Add graphic labels."
    menu := mb menu.
    submenu := (menu menuItemLabeled: 'Templates') submenu.
    (submenu menuItemAt: 1)
            labelImage: (self class oneImage).
```

```
(submenu menuItemAt: 2)
    labelImage: (self class twoImage).
(submenu menuItemAt: 3)
    labelImage: (self class threeImage).

"Set the background color."
submenu backgroundColor: ColorValue chartreuse.

^menu
```

Adding Menus to the User Interface

You add a menu to your application's GUI either by adding a menu bar to the window, adding a menu button widget, or by specifying the pop-up menu for a widget. These are all defined in the canvas and Properties Tool by specifying a menu you have defined in the Menu Editor or in a menu creating method.

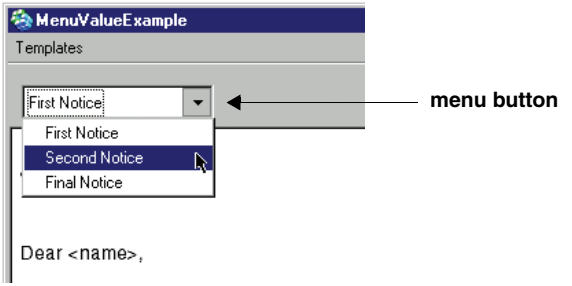
Adding a Menu Bar to a Window

A menu bar appears to the user as a set of separate menus across the top edge of a window. The menus and submenus are actually implemented by a single menu object. The menu labels displayed across the menu bar are top-level menu items in the menu object and their contents are the submenus associated with the top-level menu items.

Online example: MenuCommandExample

1. In the canvas for the window, make sure no widget is selected.
2. In a Properties Tool, turn on the **Enable** switch for the **Menu Bar** property.
3. In the **Menu** field, enter the name of the menu-creation resource method (fileMenu).
4. Create the menu definition resource method that you named in step 3, and any action methods that are invoked by the menu items.

Adding a Menu Button

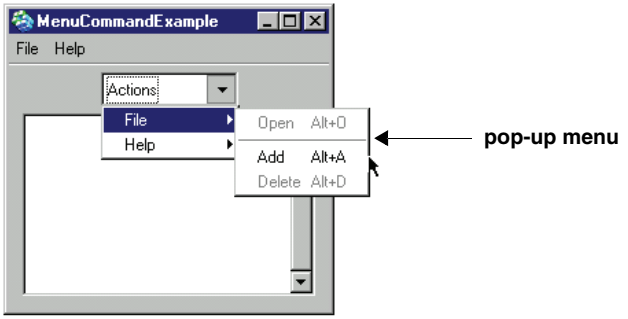


A menu button is a widget that allows you to place a drop-down menu anywhere in the canvas. Also, its label typically changes to reflect the current selection. By default, selecting an item in a button menu places the value of the menu item in the button menu widget's aspect value holder.

For information about configuing the Menu Button widget, refer to "Menu Button" in Chapter 9, "Configuring Widgets."

Online example: MenuValueExample

Adding a Popup Menu to a Widget



Several widgets, notably lists and text editors, provide a pop-up menu in response to the <Operate> mouse button.

The underlying menu is typically a menu of commands, sending the associated symbol as a message to the application model.

Online example: MenuCommandExample

1 In the **Menu** property of the widget, enter the name of the menu creation resource method (fileMenu).

2 Create the menu creation resource method (fileMenu), and any action methods named in the menu of commands.

Adding a Menu Bar or Pop-Up Menu of Values

Menu bars and pop-up menus can be used to set the value of a value holder as the result of the command they execute. You can either create a method for each message selector, defining it to set the value holder value, or enter a complex message in the menu definition. The example illustrates one way of doing the latter.

Online example: MenuValueExample

1 In the **Menu** property of the widget, enter the name of the method that returns a menu of values (templatesMenuForPopUp).

2 Use a Menu Editor or System Browser to create the menu method (templatesMenuForPopUp). In the menu, each item label is paired with a block in which the widget's aspect variable (letter) is updated with the desired value.

```
templatesMenuForPopUp
    | mb |
    mb := MenuBuilder new.

    mb
        add: 'First Notice' -> [self letter value: self class firstNotice];
        add: 'Second Notice' -> [self letter value: self class
            secondNotice];
        add: 'Final Notice' -> [self letter value: self class finalNotice].

    ^mb menu
```

Accessing Menus Programmatically

It is useful to be able to access a menu or menu items in order to modify the menu in some way. The most common use is to enable or disable (gray out) menu items.

For a menu defined using the Menu Editor, you can access an item by its **ID** property. For a menu defined either way, you can access an item by its label.

Online examples: MenuCommandExample, MenuEditorExample

In the method that is to access the menu, do the following:

1. Get the menu by sending a menuAt: message to the builder, with the name of the menu's resource method as argument.
2. Get a menu item, either by:
   - label, by sending menuItemLabeled: to the menu with the label string as the argument, or
   - name key (ID), by sending atNameKey: to the menu with the name key as a symbol as the argument.
3. Get a submenu by sending a submenu message to the menu item that is the submenu.
4. Get a submenu item by sending menuItemLabeled: or atNameKey: to the submenu, as in step 2.

MenuCommandExample accesses menu items this way to disable and enable menu items in its configureMenu method:

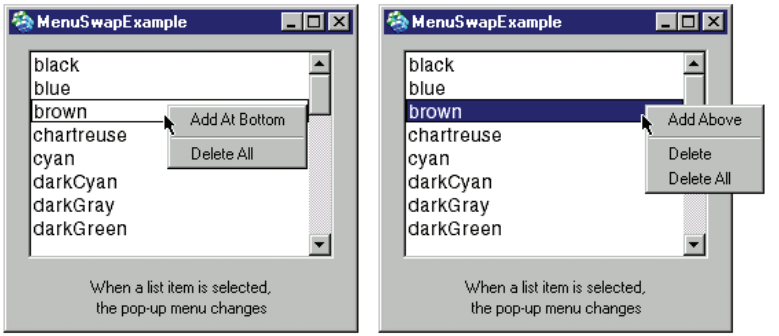
```
configureMenu
    "Disable or enable the menu items depending on whether
    a file is selected."

    | menu submenu |
    menu := self builder menuAt: #fileMenu.
    submenu := (menu menuItemLabeled: 'File') submenu.

    self files selection isNil
        ifTrue: [
            (submenu menuItemLabeled: 'Open') disable.
            (submenu menuItemLabeled: 'Delete') disable]
        ifFalse: [
            (submenu menuItemLabeled: 'Open') enable.
            (submenu menuItemLabeled: 'Delete') enable]
```

It is simple to create its interface using the Menu Editor and assign **ID** properties to the File menu and its items. The equivalent configureMenu method could then be written (assuming the ID properties are the same as the labels):

```
configureMenu
    "Disable or enable the menu items depending on whether
    a file is selected."

    | menu submenu |
    menu := self builder menuAt: #fileMenu.
    submenu := (menu atNameKey: #File) submenu.

    self files selection isNil
        ifTrue: [
            (submenu atNameKey: #Open) disable.
            (submenu atNameKey: #Delete) disable]
        ifFalse: [
            (submenu atNameKey: #Open) enable.
            (submenu atNameKey: #Delete) enable]
```

As a variation on this theme, you can operate on a collection of menu items. In MenuEditorExample, the disableDarkColors method gets the menu's collection of menu items by sending a menuItems message to the menu. It then sends a nameKey message to each menu item to obtain its name key, and disables each menu item whose name key is in the darkColors array.

Modifying a Menu Dynamically



Sometimes a menu needs to change depending on conditions within the application. The most common change is to simply disable, or "gray out" an option. More extreme, but also common, is to have different sets of menu items available depending on the state of the application.

For example, in a document editing program, there might be only options for opening documents and closing the application if no document is open, but several menus with many editing options available when a document is open.

The following sections describe several useful modifications.

Disabling a Menu Item

Applications frequently disable, or “gray out,” menu items when the selection in inappropriate for the current state. This is typically done after testing some condition in the application.

To disable a menu item, send a disable message to the menu item. Similarly, to enable a menu item, send an enable message to it.

See MenuCommandExample and the example code in the previous section.

Hiding a Menu Item

Hiding a menu item is sometimes better than disabling it. A disabled item may frustrate a user if it is not clear why it is disabled, while hiding it removes the temptation to use it. This is an interface design decision.

Online example: MenuModifyExample

1. Get the menu and menu item.

   Refer to “Accessing Menus Programmatically” above. You may need to get a submenu to get to the menu item.

2. To hide the item, send a hideItem: message to the menu, with the menu item as the argument.

3. To reveal an item, send an unhideItem: message to the menu, with the menu item as the argument.

```
adjustBenefitList
    "Hide benefit items that are not available to the currently
    selected job title."

    | bMenu item |
    bMenu := self builder menuAt: #benefitsMenu.
    item := bMenu menuItemLabeled: 'Golden Parachute'.

    "Only the President gets the Golden Parachute."
    self jobTitle value == #President
        ifTrue: [bMenu unhideItem: item]
        ifFalse: [bMenu hideItem: item].
```

Adding an Item to a Menu

You can add a new menu item to the end of a menu.

Online example: MenuModifyExample

1. Get the menu by sending a menuAt: message to the application model's builder, with the menu creation resource method as argument.
2. Send an addItemLabel:value: message to the menu. The first argument is the label string and the second argument is a command, a value, or an action block.

```
addTitle
    "Prompt for a new job title and add it to the list."

    | newTitle jMenu |
    newTitle := Dialog request: 'New title?'.
    newTitle isEmpty ifTrue: [^self].

    jMenu := self builder menuAt: #jobTitlesMenu.
    jMenu addItemLabel: newTitle value: newTitle asSymbol.

    self jobTitle value: newTitle asSymbol.
```

Removing an Item from a Menu

Online example: MenuModifyExample

1. Get the menu and the menu item to be removed.

   Refer to "Accessing Menus Programmatically" above. You may need to get a submenu to get to the menu item.
2. Send a removeItem: message to the menu, with the menu item to delete as the argument.
3. In the case of a menu button in which the current selection is displayed (that is, a menu button whose **Label** property is blank), make sure the button's value holder has a valid value.

```
deleteTitle
    "Prompt for a title and remove it from the list."

    | jMenu removableTitles title item |
    jMenu := self builder menuAt: #jobTitlesMenu.

    "Don't permit the president to be overthrown."
    removableTitles := jMenu labels
        reject: [ :nextTitle | nextTitle = 'President'].

    title := Dialog
        choose: 'Delete Title'
        fromList: removableTitles
        values: removableTitles
        lines: 8
        cancel: [^nil]
        for: Window activeController view.

    item := jMenu menuItemLabeled: title.
    jMenu removeItem: item.

    "If the deleted title is showing, pick the first title."
    self jobTitle value == title asSymbol
        ifTrue: [self jobTitle value: #President].
```

Substituting a Different Menu

The MenuSwapExample swaps menus depending on the application state. This is useful for large changes.

Online example: MenuSwapExample

1. In the **Menu** property of the widget, enter the name of a method that returns a value holder containing a menu (menuHolder).
2. In the application model, create an instance variable to hold the menu in a value holder (menuHolder).
3. Create a method (menuHolder) that returns the value of the instance variable.

```
menuHolder
    ^menuHolder
```

4. Create the starting menu (nothingSelectedMenu) and the alternate menu (colorSelectedMenu).
5. In the initialize method, get the starting menu, put it in a value holder, and assign the holder to the instance variable.

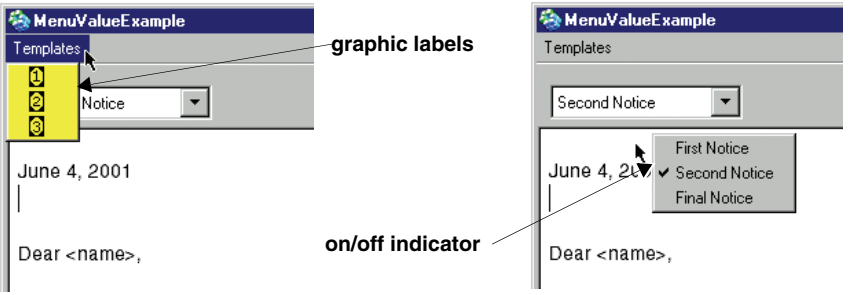
```
initialize
    colors := SelectionInList with: ColorValue constantNames.
    colors selectionIndexHolder onChangeSend: #selectionChanged to:
self.

    menuHolder := self nothingSelectedMenu asValue.
```

6 Create a method (selectionChanged) that tests to see which menu should be used and then puts the correct menu in the menu holder.

```
selectionChanged
    self colors selection isNil
        ifTrue: [self menuHolder value: self nothingSelectedMenu]
        ifFalse: [self menuHolder value: self colorSelectedMenu]
```

7 Arrange for the menu-changing method to be invoked when the relevant condition changes in the application. (In the example, the onChangeSend:to: message in the initialize method accomplishes this.)

Displaying an Icon in a Menu



Menu items can have a textual or graphical label.

Adding an Icon to a Menu

Online example: MenuValueExample

It is often useful to substitute a graphic label for a textual label or combine the two.

Send a labelImage: message to the menu item. The argument is any visual component, but typically it is a graphic image. The label string will be displaced to the right to make room for the image. The label string must have at least one character (even just a space).

```
templatesMenuForMenuBar
    | mb menu submenu |
    mb := MenuBuilder new.

    mb
        beginSubMenuLabeled: 'Templates';
        add: ' ' -> [self letter value: self class firstNotice];
        add: ' ' -> [self letter value: self class secondNotice];
        add: ' ' -> [self letter value: self class finalNotice];
        endSubMenu.

    "Add graphic labels."
    menu := mb menu.
    submenu := (menu menuItemLabeled: 'Templates') submenu.
    (submenu menuItemAt: 1)
        labelImage: (self class oneImage).
    (submenu menuItemAt: 2)
        labelImage: (self class twoImage).
    (submenu menuItemAt: 3)
        labelImage: (self class threeImage).

    "Set the background color."
    submenu backgroundColor: ColorValue chartreuse.

    ^menu
```

Displaying an On/Off Indicator

Online example: MenuValueExample

This procedure prefixes a check mark or check box to the textual label as a toggle indicator. This technique is frequently used with a menu item that represents a setting to indicate whether the condition is on or off. You can also use it to simulate a set of radio buttons in a menu, as MenuValueExample does.

1. To display an “on” indicator, send a beOn message to the menu item. The indicator is a check mark in some looks and a box in others.
2. To display an “off” indicator, send a beOff message. In some looks, beOff simply removes the “on” indicator; in others it displays a different image.

```
setCheckMark
    "In the pop-up menu, set the check box to indicate the currently
    displayed template."

    | menu item |
    menu := self builder menuAt: #templatesMenuForPopUp.

    item := menu menuItemAt: 1.
    self letter value = self class firstNotice
        ifTrue: [item beOn]
        ifFalse: [item beOff].

    item := menu menuItemAt: 2.
    self letter value = self class secondNotice
        ifTrue: [item beOn]
        ifFalse: [item beOff].

    item := menu menuItemAt: 3.
    self letter value = self class finalNotice
        ifTrue: [item beOn]
        ifFalse: [item beOff].
```

Adding a Group with a Single Indicator

Online Example: MenuSelectExample, MenuValue2Example

MenuSelectExample illustrates how to set a single menu item to show “on” and reset all others of a group of menu choices to be shown “off.” This is done by sending a beOn message to one item and sending a beOff message to all others.

MenuValue2Example is a variation in which the indication value model for each item is set programmatically for both the pop-up menu and menu button.

An alternate procedure available, which minimizes the amount of additional code, is to set the indicator in the application model. When the indication of one or more menu items is set to a value model and the model value equals the name key for the item, only that item will be shown selected among the group. Using this procedure also ensures that the “on” indication is compliant with the look preferences of the user interface for one-of-many selection. For example, for the Windows look, a dot is displayed next to the item that is “on.” This approach also matches the method used to construct a group of radio buttons (see “Radio Buttons”).

1 Open a Menu Editor by selecting **Tools → Menu Editor** in the GUI Painter Tool or **Painter → Menu Editor** in the Launcher.

2 For each menu item, enter a different identifier name in the **ID:** field and a message selector for the **Value:** field.

3 Enter the same **Indication Selector** for each menu item on the **State** page. The method you write for the selector must return a value model.

4 Choose **Menu → Install…** to install a specification for the menu in a resource method of the application model.

5 In each menu action method that you write in the application model, set the value of the indication value model to the ID of the menu item entered in step 2. For example:

```
bePoliceman
    self career value: #policeman
```

6 Set the initial value of the indication value model in your application to the menu item name you want first selected. If the initial value is nil or matches none of the items in the group, no items will be marked as "on."

Toolbars

Toolbars enable a user to select and activate application functions rapidly making their selection from a set of buttons, usually positioned below the menu bar or title of a window. Toolbar buttons use an icon to identify their functions, and serve to invoke an operation where a menu selection may be slow or tedius.

In VisualWorks, toolbars are created and modified in the same way as are menus, either using the Menu Editor or programmatically. Any menu item can correspond to a tool bar button. However, in a tool bar each item has the following additional properties:

- A label image is required for each toolbar button.
- To show a button as disabled, its image should be derived from an OpaqueImageWithDisablement instance.
- No submenus are interpreted or shown for a toolbar. Only top level items are used.
- Shortcut keys or labels defined for a toolbar button are ignored.
- An "on" state is typically shown as a depressed button, and an "off" state is shown as a raised button. In some look policies the "on" button appears highlighted.

- In addition to an icon, a help text may be associated with a tool bar button item. This help text is displayed as fly-by help when the cursor hovers over a button.
- The divider line separating a group of items in a menu appears as a space or line between buttons in a toolbar.

Currently, toolbars may only appear as buttons along the top of the window under the menu bar. On a toolbar, the first menu item corresponds to the leftmost tool bar button.

Creating a Tool Bar Image

Online example: ToolCommandExample

The source of the tool bar button graphic may be an Image created using the VisualWorks Image Editor, or created as an external bitmap file and read into VisualWorks. The graphic may be captured from the screen or edited by hand, and installed to a class side resource method in the application.

When assembling the tool bar images, note that the tool bar height will be the height of the largest image for any of the tool bar buttons. To ensure that all the tool bar buttons appear with the same size, use the same height and width for all button images. The label image for a toolbar button may be any size but the most common is between 16 to 20 pixels square. Also, there is no need to add a decorative border in the image since the tool bar button provides its own.

When you create an image for a tool bar button, the image background will be visible on the button. To avoid having the image background stand out, you can either choose the image background to be the same color as the button or create an OpaqueImageWithDisablement that masks out the background from the image. An OpaqueImageWithDisablement consists of two parts: the image and a black-and-white mask that determines the opaque and transparent regions. Given an image, you can easily create a mask from it in the Image Editor. Once you have image and mask resource methods, create another method that answers an OpaqueImageWithDisablement.

The following procedure refers to ToolCommandExample.

1. Open an Image Editor by selecting **Tools → Image Editor** in the GUI Painter Tool or **Painter → Image Editor** in the Launcher.
2. Prepare an image use as the enabled image for the toolbar button. Choose **Image → Install…** in the Image Editor and install the image as a class resource method.

3 Create and install a background mask for the button starting with the current image in the Image Editor. Install the resulting mask to your application as a class resource method.

4 Optionally, prepare and install a mask resource method to represent the inactive or disabled button. For simplicity, however, the inactive mask is usually set to be the mask created for the active button.

5 Create a new application class method that answers an OpaqueImageWithEnablement. Create the OpaqueImageWithEnablement instance by sending a figure:shape:inactiveMask: message:

```
openIcon
    ^OpaqueImageWithEnablement
        figure: self openIconImage
        shape: self openIconMask
        inactiveMask: self openIconMask
```

Adding a Toolbar

Online Example: ToolbarDemo

To use a toolbar, the following parts must be assembled in your application:

- Images for the tool bar buttons, and accessor methods for the images.
- A tool bar menu that identifies the actions and images for the tool bar.

You must also identify the tool bar menu to be shown in the application window using the GUI Painter Tool.

The following procedure refers to the ToolbarDemo example.

1 Open the Menu Editor by selecting **Tools → Menu Editor** in the GUI Painter Tool.

2 Add a menu item for a new toolbar button and enter a one word description of its function in the Label String: field. This label serves more as a mnemonic and placeholder in the menu during design than any purpose for the toolbar.

3 In each item's **Value** property, enter a message selector for the method to be performed when the item is selected.

4 On the **Details** page, enter a button image accessor class and selector in the **Label Image Class:** and **Selector:** fields, respectively. The accessor method should be a class method in the specified class.

5 Optionally, add divider lines to effectively separate items in groups.

6 Choose **Menu → Install...** to install the specification for the menu in a resource method of the application model.

7 In the GUI Painter Tool, select the Main Window icon from the tree list and, in the **Tool Bar Menu:** field, enter the selector of the menu creation resource from the step above.

8 Use the Image Editor to create a new image, and install it as a resource to the class named in step 4 above for each menu item.

Modifying a Toolbar Dynamically

Like menus, toolbars may dynamically change their enablement state, image, visibility, and indication state. A toolbar may also be swapped out for another toolbar, just as menus can. As shown in "Modifying a Menu Dynamically", this is accomplished by sending an appropriate message to a MenuItem. Unlike menus that are built and opened upon request, toolbars must also be updated to show their change of state for a MenuItem. Updating a toolbar button dynamically is simplified by assigning a value model for any of its menu item's enablement, image, visibility, or indication. When the value of the model changes its toolbar button in the application window will update visually.

Disabling a Toolbar Button

Online Example: ToolCommandExample

To show a toolbar button as disabled, the button image should be prepared as an instance of OpaqueImageWithEnablement. When a toolbar button uses such an image and its MenuItem is disabled, the button image will be shown "grayed out."

The ToolCommandExample modifies the MenuCommandExample to add a tool bar with buttons to open a selected file, add a file, and retract a file from a list. Another button on the tool bar opens the help dialog. The application disables the open or retract toolbar buttons when no file selection exists.

1 Open a Menu Editor by selecting **Tools → Menu Editor** in the GUI Painter Tool.

2 For each menu item, on the **Details** page, set the **Label Image Class:** and **Selector:** fields for the button image accessor. (Refer to "Creating a Tool Bar Image" for instructions.)

3 On the **State** page, enter the method selector in the application model that accesses a value model that determines whether the tool bar button is active (true) or disabled (false).

Hiding a Toolbar Button

Online Example: ToolCommandExample

1 Create and initialize an instance variable for a value model on a Boolean that will determine whether a toolbar button is to be shown or not.

```
initialize
    files := SelectionInList new.
    leakyTap := true.
    onHook := true.
    phoneImage := self class onHookIcon asValue.
    tapImage := self class leakyTapIcon asValue.
    hideHook := false asValue.
    hideTap := false asValue.
```

2 In the postBuildWith: method of your application model, access the toolbar menu from the builder.

3 For any toolbar button that is to be dynamically hidden, access its menu item and send the hidden: message with its value model as argument.

```
postBuildWith: aBuilder
    | toolMenu |
    toolMenu := aBuilder menuAt: #toolBar.
    (toolMenu atNameKey: #connection)
        labelImage: self phoneImage;
        hidden: self hideHook.
    (toolMenu atNameKey: #issues)
        labelImage: self tapImage;
        hidden: self hideTap.
```

4 When you wish to hide a toolbar button, change its hidden value to true. To show the toolbar button, change its model value to false.

```
toggleHideHook
    self hideHook value: self hideHook value not
```

Changing a Toolbar Button Image

Online Example: ToolCommandExample

1 Create and initialize an instance variable for a value model on the button image:

```
initialize
    files := SelectionInList new.
    leakyTap := true.
    onHook := true.
    phoneImage := self class onHookIcon asValue.
    tapImage := self class leakyTapIcon asValue.
    hideHook := false asValue.
    hideTap := false asValue.
```

2 In the postBuildWith: method of your application model, access the toolbar menu from the builder.

3 For any button image that is to be dynamically changed, access its menu item and send the labelImage: message with its value model as argument.

```
postBuildWith: aBuilder
    | toolMenu |
    toolMenu := aBuilder menuAt: #toolBar."step 2"
    (toolMenu atNameKey: #connection)
        labelImage: self phoneImage;"step 3"
        hidden: self hideHook.
    (toolMenu atNameKey: #issues)
        labelImage: self tapImage;"step 3"
        hidden: self hideTap.
```

4 Change the button labelImage model value to a new image when the button image is to be changed.

```
beOffHook
    | menuBar|
    menuBar := self builder menuAt: #menuBar.
    onHook := false.
    (menuBar atNameKey: #onHook) hidden: false.
    (menuBar atNameKey: #offHook) hidden: true.
    self phoneImage value: self class offHookIcon."step 4"
    ((self builder componentAt: #toolBar) componentAt: #connection)
widget helpText: 'Disconnect'
```

Displaying an On/Off Indication

In some applications it is desirable to have a "locking" toolbar button that stays depressed when on.

Online Example: UniversalSelectExample

1 Assign an instance variable to a value model for a button's indication state. Set the initial value to true if the button is to display "on," or false to display "off."

2 In the Menu Editor, set the **Indication Selector:** field to the accessor selector for the value model above. Or, programmatically, send an indication: message to the MenuItem with the value model as argument.

3 To change the toolbar button to "on," set the value of the model above to true.

Adding a Group of Buttons with a Single Selection

Online Example: UniversalSelectExample

As in section "Adding a Group with a Single Indicator", create a group of menu items that each identify a different name key, but that all reference the same value model for indication. When the value of the model is set to the name key of one item in the group its toolbar button appears "on" and all others appear "off."

Extending Menus and Toolbars

Online examples: MenuPragma1Example, MenuPragma2Example

VisualWorks tools are designed to allow you to easily extend their menus and tool bars. This is done by adding methods that invoke a special menu item definition pragma. This is useful especially for developers who are enhancing the standard tool set, or adding new tools.

The menu pragma is a special syntactical element that can be placed in a method definition:

```
< pragmaName: ...
    >
```

where ... is a series of menu property specifications. The set of pragmas is defined in Menu class method pragmas. Use any of these selectors to define a new menu or tool bar element.

Within VisualWorks, extending menus is done by a variety of system parcels. For example, the UIPainter parcel adds menus and tool buttons to the Visual Launcher when it is loaded. Using the system browser to browse the #actions category in the VisualLauncher class, you will find methods like:

```
browseApplications
    "Open a new UIFinder."
    <menuItem: '&Resources'
        icon: #finderIcon
        nameKey: nil
        menu: #(#menuBar browse)
        position: 10.01>
    <menuItem: 'Browse Applications'
        icon: #finderIcon
        nameKey: nil
        menu: #(#launcherToolBar)
        position: 20.02>

    self openApplicationForClassNamed: #UIFinderVW2
```

The first pragma adds the Resources action to the Browse menu (#menuBar browse) on the Launcher's menu bar. The second pragma adds the finder button to the Launcher's tool bar (#launcherToolBar). Both assign the named icon, but neither assigns a nameKey, which would be used to access the item programmatically.

Following any menu item pragma label, the pragma specifies the actions to be performed when the item is selected, as a normal Smalltalk expression.

Pragma Parameters

The following parameters occur in the menu pragmas:

menuItem: *aStringOrUserMessage*
: The menu item label as either a String or UserMessage. As a UserMessage the argument should be an array specifying the message key, the catalog ID (optional), and default message string. For example, #(#key #catalog 'default') or #(#key 'default'). If the string includes an ampersand (&), the following character becomes a keyboard access character for the menu item and appears underlined in the menu label.

submenu: *aStringOrUserMessage*
: A submenu menu item heading as either a String or UserMessage. If you intend the label to be a UserMessage, the argument should be an array to specify the message key, the catalog ID (optional), and default string. For example, #(#key #catalog 'default') or #(#key 'default').

nameKey: *aSymbolOrNil*
Sets the symbol used to identify this menu item, or nil if not needed. This symbol is used to access the menu item and may be used to change its state programmatically, such as to disable or enable it. If the menu item is a submenu, its nameKey would appear in each of its item menu pragmas in the menu: parameter array.

icon: *aSelector*
Sets the accessor selector for the icon for the menu item, if used. The accessor method is expected to be defined among class methods.

menu: *anAspectArray*
Sets an array of selectors naming the aspect path to reach the menu item for this pragma, starting from the parent menu. The first array element is the accessor selector for the parent menu or tool bar. The remaining elements are the nameKeys of each submenu following under the parent to reach the menu item. For example, if a menu pragma defines an item under submenu **Advanced** of the VisualLauncher **Tools** menu, its aspect array would appear as

```
#(#menuBar #tools #advanced)
```

position: *aFloat*
Specifies the insertion position, represented as a Float. The integer part represents the group, which is a part of the menu separated by a line. The fractional part represents the menu item's position in the group. If the group number identifies a group that doesn't already exist, a new group is added. For more information, see "Setting the Menu Item Position".

enablement: *aSymbolOrNil*
Sets the accessor selector used to determine whether this menu item is enabled, or nil if the menu item is enabled always. The method must answer a Boolean: true if enabled, or false if disabled.

indication: *aSymbolOrNil*
Sets the accessor selector used to determine whether this menu item has a check selection, or nil if the menu item never checked. The method must answer a Boolean: true if checked, or false if not checked.

shortcutKeyCharacter: *aCharacterOrSymbol*
Specifies the shortcut key for activating this choice from the keyboard. The entry is the character of the key, unless the key is a function key (e.g., F1) in which case the entry is a symbol. When specified, the menu item will display with the key combination to the right of its label.

shortcutModifiers: *anInteger*

A code indicating whether any combination of a control, alt, or shift key is required with the shortcut key. The shift, control, and alt keys are each given the index 1, 2, and 8, respectfully. The shortcut modifier code is the sum of all these three key codes pressed. The following table summarizes the possible code combinations by key press.

Shift	Ctrl	Alt	Code
			0
✓			1
	✓		2
		✓	8
✓	✓		3
✓		✓	9
	✓	✓	10
✓	✓	✓	11

helpText: *aStringOrUserMessage*

Specifies the String or UserMessage to appear as fly-by help message. Applies only to tool bar items. As a UserMessage the argument should be an array with the message key, the catalog ID (optional), and default message string, for example #(#key #catalog 'default') or #(#key 'default').

Menu Pragma Forms

The following is a list of all pragma forms. Following that, selective examples are provided. Pragma forms are defined in class MenuAutomaticGenerator.

Shortcut	Enable	Indicator	Icon	Help	Pragma
					<menuItem: aStringOrUserMessage nameKey: aSymbolOrNil menu: anAspectArray position: aFloat>
✓					<menuItem: aStringOrUserMessage nameKey: aSymbolOrNil shortcutKeyCharacter: aCharacterOrSymbol shortcutModifiers: anInteger menu: anAspectArray position: aFloat>
	✓	✓			<menuItem: aStringOrUserMessage nameKey: aSymbolOrNil enablement: aSelectorOrNil indication: aSelectorOrNil menu: anAspectArray position: aFloat>
✓	✓	✓			<menuItem: aStringOrUserMessage nameKey: aSymbolOrNil enablement: aSelectorOrNil indication: aSelectorOrNil shortcutKeyCharacter: aCharacterOrSymbol shortcutModifiers: anInteger menu: anAspectArray position: aFloat>
			✓		<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil menu: anAspectArray position: aFloat>
	✓	✓	✓		<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil enablement: aSelectorOrNil indication: aSelectorOrNil menu: anAspectArray position: aFloat>

Shortcut	Enable	Indicator	Icon	Help	Pragma
			✓	✓	<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil menu: anAspectArray position: aFloat helpText: aStringOrUserMessage>
✓			✓		<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil shortcutKeyCharacter: aCharacterOrSymbol shortcutModifiers: anInteger menu: anAspectArray position: aFloat>
✓	✓	✓	✓		<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil enablement: aSelectorOrNil indication: aSelectorOrNil shortcutKeyCharacter: aCharacterOrSymbol shortcutModifiers: anInteger menu: anAspectArray position: aFloat>
	✓	✓	✓	✓	<menuItem: aStringOrUserMessage icon: aSelector nameKey: aSymbolOrNil enablement: aSelectorOrNil indication: aSelectorOrNil menu: anAspectArray position: aFloat helpText: aStringOrUserMessage> <submenu: aStringOrUserMessage nameKey: aSymbolOrNil menu: anAspectArray position: aFloat>
	✓				<submenu: aStringOrUserMessage nameKey: aSymbolOrNil enablement: aSelectorOrNil menu: anAspectArray position: aFloat> <computedSubmenu: aStringOrUserMessage nameKey: aSymbolOrNil menu: anAspectArray position: aFloat>

Minimal Menu Pragma

The minimal of menu pragmas is:

```
<menuItem: aStringOrUserMessage
    nameKey: aSymbolOrNil
    menu: anAspectArray
    position: aFloat>
```

Since the pragma lacks an icon definition, this menu item cannot be used in a tool bar.

```
<menuItem: 'Inspect instances..'
    nameKey: nil
    menu: #(#menuBar #help #tricks)
    position: 20.01>
```

This pragma inserts a menu item labeled “Inspect instances..” in the first position (.01) of the second (20) menu group. This item would be accessed by selecting **Help** (menu item nameKey #help) from the menu bar, and then selecting the **Stupid Menu Tricks** submenu (menu item #tricks), which is added as shown below under “Submenu pragmas”. No provision is being made for changing the menu item programmatically, so the nameKey: parameter is set to nil.

The specific values required to ensure this menu item position may differ depending on the position numbers of the current menu items. For groups, incrementing by 10 allows room for inserting another group if necessary, for example by using a value of 15.

Menu Label as a UserMessage

Instead of providing a literal String for the menu item, you can provide a 2- or 3-element Array which is used to produce a UserMessage. Using a UserMessage if preferred especially for applications that require localization. For more information about UserMessages and catalogs, refer to the *Internationalization Guide*.

The label is looked up by key in a catalog of user strings. If no catalog entry is found by that key, the default String is used.

The 2-element array specifies only the key, as a Symbol literal, and default string, for example:

```
#(#lookupKey 'Inspect instances..')
```

The 3-element array specifies the catalog, also as a Symbol literal, for example:

```
#(#lookupKey #menuCatalog 'Inspect instances..')
```

If a catalog is specified, only that catalog is searched for the key. If no catalog is specified, all catalogs are searched.

Using the 3-element form, the previous example might become:

```
<menuItem: #(#lookupKey #menuCatalog 'Inspect instances')
    nameKey: nil
    menu: #(#menuBar #help #tricks)
    position: 20.01>
```

Including a Shortcut Key

```
<menuItem: aStringOrUserMessage
    nameKey: aSymbolOrNil
    shortcutKeyCharacter: aCharacterOrSymbol
    shortcutModifiers: anInteger
    menu: anAspectArray
    position: aFloat>
```

This pragma adds a shortcut to perform the menu action by pressing the key specified by shortcutKeyCharacter: in combination with the control keys specified by shortcutModifiers: from the keyboard. For example,

```
<menuItem:'Topics'
    nameKey: nil
    shortcutKeyCharacter: #F1
    shortcutModifiers: 0
    menu: #(#menuBar #help)
    position: 10.0>
```

In this case, the F1 key, without modifiers (Ctrl, Alt, Cmd, Shift), invokes the menu item labeled **Topics** on the **Help** menu, which is added to the first position of the first menu group. “F1” is shown to the right of the menu item, indicating that it is the shortcut key combination.

The shortcutModifiers: parameter is an integer indicating a combination of Ctrl, Alt, and Shift modifier keys, or none. A table giving the values is provided above, under “Pragma Parameters”. To modify the above example to invoke the item by pressing Alt+T, change it to:

```
<menuItem:'&Topics'
    nameKey: nil
    shortcutKeyCharacter: $T
    shortcutModifiers: 8
    menu: #(#menuBar #help)
    position: 10.0>
```

The “8” indicates the Alt key modifier.

The “&” in the label string adds a further decoration, underlining the following letter. (To show an & in the menu item, enter &&.) If you open the menu by clicking on it, you can then select that menu item by pressing the underlined character key. For good GUI design, this should be the same character as specified as the shortcutKeyCharacter:.

By some conventions, such as on Windows, this also indicates an <Alt>, <T> sequence, rather than an <Alt>+<T> combination. However, the <Alt> and <Ctrl> keys alone have no effect at this time in VisualWorks, so the sequence is not supported.

Add Enablement and Selection Indicators

```
<menuItem: aStringOrUserMessage
    nameKey: aSymbolOrNil
    enablement: aSelectorOrNil
    indication: aSelectorOrNil
    menu: anAspectArray
    position: aFloat>
```

The enablement: and indication: keywords occur together in menu pragmas, though either or both can be set to nil. The enablement: parameter occurs alone in submenus pragmas, since an indicator would be inappropriate.

The enablement: parameter sets the accessor method selector that answers whether the menu item should be enabled; or nil if the item is always enabled. Similarly, the indication: parameter sets the accessor method selector that answers whether the menu item has an indicator mark; or nil if the item is never marked. For example,

```
<menuItem: '&Delete'
    nameKey: nil
    enablement: #anyFileSelected
    indication: nil
    menu: #(#menuBar #file)
    position: 20.04>
```

This inserts a menu item labeled **Delete**, which is enabled if #anyFileSelected answers true (e.g., if a file is selected in the application). There is no indicator mark if this item is selected.

```
<menuItem: '&Toolbar'
    nameKey: nil
    enablement: nil
    indication: #showsToolbar
    menu: #(#menuBar #view)
    position:10.02>
```

This inserts a menu item labeled **Toolbar**, which shows the item marked if #showsToolbar answers true (e.g., if a tool bar is shown by the application). The menu item is always enabled.

Adding an Icon

```
<menuItem: aStringOrUserMessage
    icon: aSelector
    nameKey: aSymbolOrNil
    menu: anAspectArray
    position: aFloat>
```

The pragmas with the icon: keyword add an icon to the menu item. If the item appears in a menu, the icon is displayed to the left of the label. If the item appears in a tool bar, the icon is the button image. For example,

```
<menuItem: '&Ring bell'
    icon: #bellIcon
    nameKey: nil
    menu: #(#menuBar #help #tricks)
    position: 10.01>
```

This pragma adds a **Ring bell** menu item and icon from accessor bellIcon to the **Menu tricks** submenu (menu item #tricks) of **Help** (menu item #help).

```
<menuItem: 'Time'
    icon: #timeIcon
    nameKey: nil
    menu: #(#launcherToolBar)
    position: 20.03>
```

This pragma inserts a menu button in the tool bar named launcherToolBar, using class icon accessor message timeIcon.

Adding Fly-by Help

```
<menuItem: aStringOrUserMessage
    icon: aSelector
    nameKey: aSymbolOrNil
    menu: anAspectArray
    position: aFloat
    helpText: aStringOrUserMessage>
```

The helpText: pragma keyword adds a fly-by help message to a tool bar item. For example,

```
<menuItem: '&Time'
    icon:#timeIcon
    nameKey: nil
    menu: #(#launcherToolBar)
    position: 20.03
    helpText: 'Time now'>
```

As with the menuItem: parameter, the help text can be provided by as a UserMessage, which is especially useful when an application needs to be localized:

```
<menuItem: #(#timeItem #menusCatalog '&Time')
    icon:#timeIcon
    nameKey: nil
    menu: #(#launcherToolBar)
    position: 20.03
    helpText: #(#timeHelp #helpCatalog 'Time now')>
```

Submenu pragmas

Two pragmas are provided for adding submenus. The simplest is:

```
<submenu: aStringOrUserMessage
    nameKey: aSymbolOrNil
    menu: anAspectArray
    position: aFloat>
```

This adds a simple submenu according to the given declaration. For example,

```
<submenu: 'Stupid Menu Tricks'
    nameKey: #tricks
    menu: #(#menuBar #help)
    position: 20.01>
```

adds a submenu labeled **Stupid Menu Tricks** to the **Help** (menu item #help) menu intended for the second group at position one.

The only enhancement support of submenus is the addition of enablement, which is done with this pragma:

```
<submenu: aStringOrUserMessage
    nameKey: aSymbolOrNil
    enablement: aSelectorOrNil
    menu: anAspectArray
    position: aFloat>
```

The enablement: keyword takes the usual message selector argument:

```
<submenu: 'Stupid Menu Tricks'
    nameKey: #tricks
    enablement: #tricksAvailable
    menu: #(#menuBar #help)
    position: 20.01>
```

Since a submenu does not associate an action with its selection any Smalltalk code appearing below the pragma is not normally invoked by the menu.

Computed Submenu Pragma

Online Example: MenuPragma2Example

The computed submenu pragma is used to add a submenu that is defined programmatically in your application.

```
<computedSubmenu: aStringOrUserMessage
    nameKey: aSymbolOrNil
    menu: anAspectArray
    position: aFloat>
```

Adds the menu answered by the method that contains this pragma as a submenu. For example:

```
emphasisChoices
    "Emphasize the text selected with an emphasis from this menu"
    <computedSubmenu:'Emphasis'
        nameKey: #text
        menu: #(menuBar )
        position:10.02>

    | mb |
    mb := MenuBuilder new.
    mb add: 'Plain' -> [textStyle value: #plain];
        add: 'Bold' -> [textStyle value: #bold];
        add: 'Italic' -> [textStyle value:#italic];
        add: 'Underlined' -> [textStyle value:#underline].
    ^mb menu
```

Adding Items to an Application's Menu or Tool Bar

A menu or tool bar in an application is not updated with menu pragmas until requested by sending the message augmentFrom:to:menuName: or augmentFrom:to:menuName:for: to the menu. These messages search a section of a class heirarchy, specified by a start class and a stop class, for menu pragmas.

To augment a menu from menu pragmas named in class hierarchy, send a message of the form:

```
aMenu augmentFrom: startClass
    to: stopClass
    menuName: menuName
    for: definer
```

where:

startClass
Either *stopClass* or a subclass of *stopClass* that will be first examined for menu pragmas.

stopClass
Either *startClass* or a superclass of *startClass* that will be last examined for menu pragmas.

menuName
The menu ID symbol of the menu to be built. Only menu pragmas whose first element of their menu: access array match this symbol are augmented to the menu receiver.

definer
An object that performs any methods with pragma computedSubmenu:nameKey:menu:position: to generate a menu to add to the receiver. If nil, ignore menu pragmas that include computedSubmenu:nameKey:menu:position:.

For example, to augment an application's menu bar named menuBar for pragmas appearing in its class its postBuildWith: method might appear as

```
postBuildWith: aBuilder

    (aBuilder menuAt: #menuBar)
        augmentFrom: self class
        to: self class
        menuName: #menuBar
        for: self.
```

Setting the Menu Item Position

The position: argument specifies the location of the menu or toobar item as a floating-point constant that sets the ordering and grouping of items.

Menu items are originally numbered, so the first group is numbered 10 and successor groups increment by 10. Similarly, the first item in a group is at position .01, and the count ascends by 0.01. So a menu with two groups of three items each has its items numbered 10.01, 10.02, 10.03, 20.01, 20.02, 20.03.

To define a new group, assign a new integer part. For example, using 15 would insert a new group between the two existing groups.

To insert a new item between existing items, use a fractional part. For example, using 10.025 would insert a new menu item between the existing second and third items. Using 10.04 would append an item to the group.

Adding Items to the Launcher

As illustrated above, methods for adding menu and toolbar items to the VisualLauncher are added to the VisualLauncher class, in the actions protocol. The launcher updates its button displays immediately and automatically.

For adding a menu item, specify that the change is in the menu bar, and which submenu the addition goes in, in an array:

```
menu: #(#menuBar browse)
```

Here, #menuBar specifies the method name that defines the menu bar, and browse is the name key of the Browse menu. Examine menuBar method for other name keys.

For adding a tool bar button, specify the class method that returns the appropriate icon. This may be an image resource method, but may be an intermediate method. For example, the Resource Finder method tests for the color depth, and returns the resource method for the appropriate icon:

```
finderIcon
    ^Screen default colorDepth == 1
        ifTrue: [self BWAppFinder24]
        ifFalse: [self CGAppFinder24]
```

Adding Items to a Browser

A similar approach is used for extending the menu bar operations and <Operate> menus corresponding to specific panes in browsers. For each pane in the browsers, there is a subclass of BrowserHelper, in which you can make any required additions. For example, the ClassesBrowserHelper class defines various features of a classes pane.

By simply adding a method specifying a menu pragma (in the actions protocol), you can add an item to the menu bar menu and to the <Operate> menu for the pane controlled by that help class. For example, add this method to ClassesBrowserHelper, and test the change to a new instance of a browser with a classes pane:

```
myMenuItem
    <menuItem: 'My new action'
        nameKey: nil
        menu: #(#listMenu)
        position 40.1>
```

Since no icons are involved, this shorter version of the menu item pragma can be used.

Menu and Toolbar Events

Unlike other triggered events sent by the system, the menu and tool bar events are the only ones that pass along arguments. In all cases, the argument is the ID (also known as the menuKey) of the menu, toolbar button or menu item. If there is no ID for the menu, toolbar button or menu item then nil is sent.

#menuOpened:
When a main menu item is opened because of navigation by the keyboard or the mouse, the menu bar triggers the #menuOpened: event. The ID (menuKey) of the menu is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

#menuClosed:
When a main menu item is closed because of navigation by the keyboard or the mouse, or a menu item being selected, the menu bar triggers the #menuOpened: event. The ID (menuKey) of the menu is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

#menuItemSelected:
When any menu item is selected, without regard to if the menu item is on a main menu or a sub-menu off of one of the main menus, the menu bar triggers the #menuItemSelected: event. If the menu item has no action associated with it, then this event is not triggered. The ID (menuKey) of the menu item is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

#submenuOpened:
When a menu item has a sub-menu, and that sub-menu is opened because of navigation by the keyboard or mouse, the menu bar triggers the #submenuOpened: event. The ID (menuKey) of the menu is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

#submenuClosed:

When a menu item has a sub-menu, and that sub-menu is closed because of navigation by the keyboard or mouse, or a sub-menu item being selected, the menu bar triggers the #submenuOpened: event. The ID (menuKey) of the menu is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

Popup Menu Events

Individual widgets are allowed to have local popup / operate menus. These events are triggered by the underlying widget when one of those popup menus is created, and if one of the menu items is selected.

#popupMenuCreated

When a widget creates a popup (<Operate>) menu, the widget triggers the #popupMenuCreated event.

#popupMenuItemSelected:

When a menu item on a popup menu is selected, the widget triggers the #popupMenuItemSelected: event. If the menu item has no action associated with it, then this event is not triggered. The ID (menuKey) of the menu item is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

Toolbar Events

Toolbars have a simple event interface, that only triggers when one of the tool bar buttons is pressed.

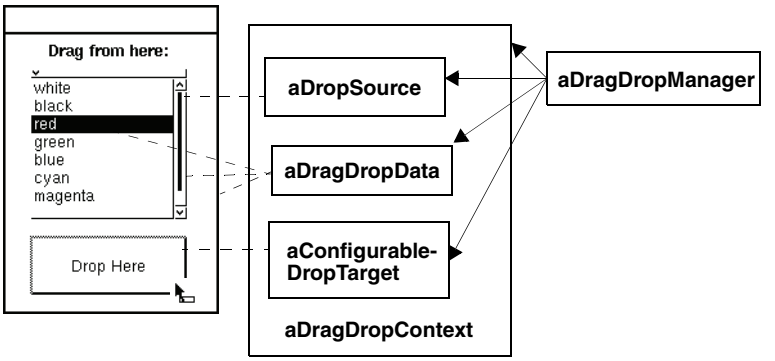
#toolBarButtonSelected:

When a toolbar button is pressed, the tool bar triggers the #toolBarButtonSelected: event. If the toolbar button has no action associated with it, then this event is not triggered. The ID (menuKey) of the toolbar button is sent as a parameter of this event if it is defined. If no ID (menuKey) is defined, then nil is sent as the parameter.

6

Drag and Drop

About Drag and Drop



Drag and drop technology allows a user to select an object using the mouse pointer, drag that object to another location on the screen, and drop it. This is useful, for instance, for moving or copying items from one list to another. This operation is common in some environments for moving files (items) from one directory (a list) to another.

In VisualWorks, you implement drag and drop by setting one or more widgets as *drop sources* and others as *drop targets*. Currently, only list widgets can be a drop source, while a drop target can be any widget except a linked or embedded dataform. Drop sources and drop targets may be in the same interface, or they may be in the interfaces of different applications.

You set up drop sources and drop targets by specifying various message names in their properties, and then programming the relevant application model(s) to respond to these messages.

Drag and Drop Framework Classes

In a running application, a drag-and-drop interaction is carried out by instances of several framework classes. Some of these instances are created as a result of the code you write, while others are created automatically when the interface is built or when drag and drop is underway. Each drag-and-drop interaction involves an instance of these classes:

DragDropData
: Holds the data to be transferred, plus information about where the drag originated (the widget's controller, containing window, and application model).

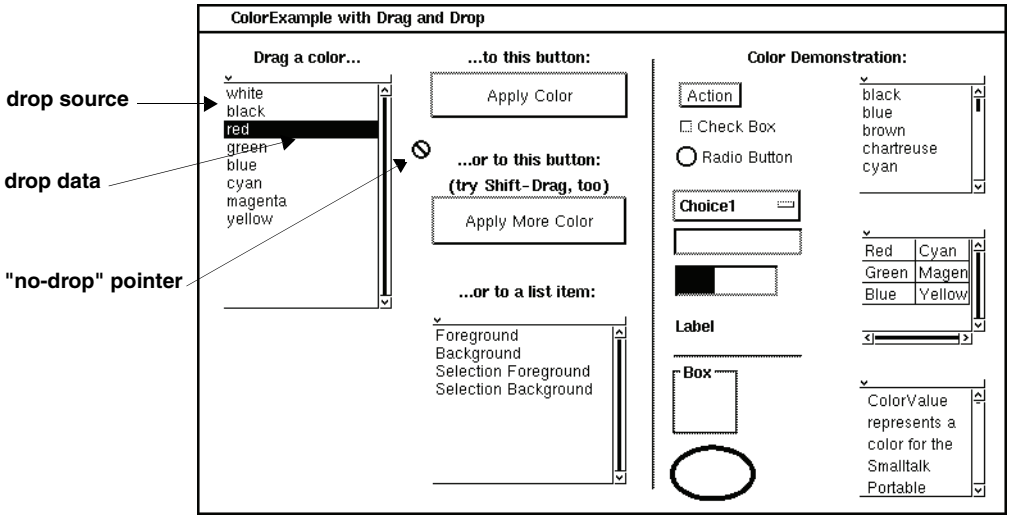
DropSource
: Defines the shapes that the pointer can have during a drag, based on the drop source. The default pointer shapes indicates whether a move, a copy, or no transfer would take place.

DragDropManager
: Tracks the mouse pointer throughout the drag, and is responsible for setting the pointer's shape based on location. When a drop occurs in a drop target, the DragDropManager sends a message to process the transferred data.

ConfigurableDropTarget
: Instances identify the widgets that are potential drop targets.

DragDropContext
: Combines the DragDropData, the DropSource, and the ConfigurableDropTarget in a convenient object for passing as an argument in various messages.

Adding a Drop Source



A drop source is a widget from which a drag can originate, and holding the data to be transferred. Currently, only the List and Table widgets can serve as a drop source.

You enable a list as a drop source by providing values for the **Drag OK** and **Drag Start** properties on the **Drop Source** page of the Properties Tool. These properties specify the messages that the widget sends when the user starts a drag operation, by pressing a mouse button and moving the mouse.

You program the widget's application model to respond to these messages as follows:

- The drag-ok method must return a Boolean to indicate whether drag and drop is appropriate from this drop source.
- The drag-start method creates DragDropData, DropSource, and DragDropManager instances.

Online example: ColorDDExample

1 Add a list widget to the canvas, set its **Aspect** and **ID** properties, install the canvas and define its aspect instance variable and accessor method, and initialize the SelectionInList, as usual for a list (refer to "Lists" in Chapter 9, "Configuring Widgets").

2 In the list's **Drag OK** property, enter the name of the method that will determine whether a drag can proceed from this list (colorWantToDrag:).

The selector must end with a colon, because the widget sends this message with its controller as argument. This is true even if the controller isn't used.

3 In the list's **Drag Start** property, enter the name of the method that will initiate drag and drop (colorDrag:)

This selector must also end with a colon, for the same reason.

4 Leave the **Select On Down** property selected.

This causes a selection to occur when the mouse is pressed down to start the drag, which is the normal drag and drop behavior, rather than waiting for the mouse to be released. Apply the properties and install the canvas.

5 Create the drag-ok method (colorWantToDrag:).

This method must return a Boolean (true to permit drag and drop, false to prevent it), and must accept a Controller as its argument, even if that controller isn't used.

```
colorWantToDrag: aController
"Determine whether to permit a drag to start from this widget. In
 this case, make sure that there is data to drag and that the drag
starts a selection."

    ^self color list size > 0 and: [self color selection notNil]
```

6 Create the drag-start method (colorDrag:) in the drag source protocol.

This method also must accept a Controller as its argument. In the method:

- Create a DragDropData instance.
- Send a key: message to the DragDropData instance to specify a symbol (#colorChoice) that identifies the kind of data being stored. (A drop target can use this key to filter out inappropriate kinds of data).
- Send messages to the DragDropData instance to specify any further information that a drop target might use when evaluating this drag. Typically, you send a contextWindow: message specifying the containing window, a contextWidget: message specifying the list widget, and a contextApplication: message

specifying the application model.

- Send a clientData: message to the DragDropData instance to store the object to be transferred (in the example, the color that is currently selected). Notice that the color choice is stored in an IdentityDictionary, which is a general technique for storing multiple related pieces of data.
- Create an instance of DropSource to make predefined kinds of visual feedback available during the drag.
- Create an instance of DragDropManager and initialize it with the DropSource and DragDropData instances.
- Send a doDragDrop message to the DragDropManager to start the drag and drop.

```
colorDrag: aController
"Drag the currently selected color. Provide all available
information about the context of the color so that the drop
target can use whatever it needs."

    | ds dm data |
    data := DragDropData new.
    data key: #colorChoice.
    data contextWindow: self mainWindow.
    data contextWidget: aController view.
    data contextApplication: self.
    data clientData: IdentityDictionary new.
    data clientData at: #colorChoice put: self color selection.

    ds := DropSource new.

    dm := DragDropManager
            withDropSource: ds
            withData: data.
    dm doDragDrop
```

Note that when the drag and drop completes, the doDragDrop message returns a symbol which can be stored in a temporary variable and then used to trigger further actions (such as cutting the dragged data out of the drop source list). This symbol is ignored in the colorDrag: method.

Dragging Multiple Selections

A multiple-selection list can be the source of several drag options. You set up a multiple-selection list the same way as a single-selection list, except that:

1 In the Properties Tool for the List widget, click the list's **Multi Select** property.

2 Initialize the list's aspect variable with a MultiSelectionInList instead of a SelectionInList.

3 Send the selections message (instead of selection) to obtain the selected data to store in the DragDropData instance.

   The selections message returns an ordered collection of objects.

Adding a Drop Target

A drop target is a widget in which data can potentially be dropped. You can set up any window or widget as a drop target except a linked or embedded dataform.

In general, you set up a drop target by filling in one or more of its properties on the **Drop Target** page of the Properties Tool, and then implementing corresponding methods in the application model. Each of a widget's **DropTarget** properties specifies the name of a message that the DragDropManager sends at various points after the drag encounters this widget.

At a minimum, you must fill in the widget's **Drop** property to specify the name of a method that implements the desired response when a drop occurs in that widget. Filling in the **Drop** property causes the builder to set up the widget with a ConfigurableDropTarget instance so that the DragDropManager can recognize the widget as a drop target.

In addition, you normally fill in the widget's **Entry**, **Over**, and **Exit** properties to specify the names of methods that provide visual feedback when the pointer is dragged across the widget. Typically, these methods specify the pointer's shape and adjust the drop target's appearance to signal whether the drop target can accept a drop from this particular drag. Strictly speaking, these properties are optional, in that drag and drop can function without them. However, providing visual feedback is normally required by user interface design guidelines.

Providing Visual Feedback During a Drag

As part of adding a drop target, you normally arrange for visual feedback to be given to users when they drag the pointer over it. The purpose of this feedback is to let users know whether a drop can be accepted from this particular drag, and, if so, what kind of transfer may result. Visual feedback typically includes changing the pointer's shape and adjusting the drop target's appearance—for example, by highlighting a button, changing a label, or scrolling a list to track the pointer's movement over it.

Drop Target Messages

You arrange for visual feedback by filling in the drop target's **Entry**, **Over**, and **Exit** properties with the names of messages to be sent by the DragDropManager at various points during a drag. You then implement methods in the application model to respond to these messages:

- The entry message is sent as soon as the pointer enters the widget's bounds. The method typically saves the drop target's visual state for restoring later, and/or toggles simple visual characteristics such as highlighting.
- The over message is sent immediately after the entry message, and then every time the pointer moves within the widget's bounds. The method typically adjusts the drop target's appearance in response to pointer location or a modifier key press.
- The exit message is sent wherever the pointer is dragged out of the widget's bounds before the mouse button is released. The method typically restores the widget's original appearance.

Each method has access to the dragged data through the DragDropContext instance that is passed to it by the DragDropManager. The methods query the DragDropContext to decide what kind of visual feedback to provide. Furthermore, these methods use the DragDropContext to save and restore drop target characteristics.

Note that no changes are made to a drop target's appearance unless you implement them in these methods. Programmatic techniques for changing widget appearance are described in Chapter 9, "Configuring Widgets".

Pointer Shapes

Each of the entry, over, and exit methods control the pointer shape by returning an *effect symbol*. The DragDropManager passes each effect symbol to the operation's DropSource instance, which sets the pointer shape accordingly. A standard DropSource recognizes these basic effect symbols:

#dropEffectNone
: Produces a pointer shaped like a circle with a slash through it; usually indicates that no transfer is possible in the pointer's current location.

#dropEffectMove
: Produces an arrow-shaped pointer with an open box below it; usually indicates a simple transfer such as a move (data is cut from the source after the transfer).

#dropEffectCopy
: Produces the same pointer as #dropEffectMove, but with a plus sign; usually indicates a modified transfer such as a copy (data is left in the source after the transfer).

Changing Color During a Drag

Online example: ColorDDExample

This example highlights the **Apply Color** button and changes the pointer's shape while the pointer is in the button.

1. In the canvas, select the **Apply Color** button, and set its **ID** property (in this case, enter #applyColorButton).
2. On the **Drop Target** page of a Properties Tool, fill in the widget's **Entry**, **Over**, and **Exit** properties with the names of the messages to be sent during the drag (**applyColorEnter:**, **applyColorOver:**, and **applyColorExit:**, respectively). Each selector must end with a colon. Apply the properties and install the canvas.
3. In a System Browser, add an entry method (applyColorEnter:) in an appropriate protocol (in this case, drop target - button1). The method must accept a DragDropContext instance as an argument.
4. In the entry method, test the dragged data to determine what kind of feedback to provide (positive feedback if the data is a color choice, and negative feedback otherwise). Send a key message to the DragDropContext instance to obtain the identifying symbol that was assigned when the drag started. If the data's key is not #colorChoice, return an effect symbol (#dropEffectNone) to signal that a drop is not allowed.

5 If the dragged data is acceptable, highlight the button as if it were pressed and return an effect symbol (#dropEffectMove) that signals permission to drop.

```
applyColorEnter: aDragContext
"A drag has entered the bounds of the Apply Color button. Test whether a drop
    would be permitted here with this data. If so, cause the button to be
    highlighted as if it were pressed, and return a symbol that indicates the
    feedback to be given to the user."

    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    (self widgetAt: #applyColorButton)
        isInTransition: true.

    ^#dropEffectMove.
```

6 Add an over method (applyColorOver:) that accepts a DragDropContext instance as an argument.

7 In the over method, test the dragged data and return the appropriate effect symbols. No other processing is necessary in this method because the button's highlighting does not vary with the pointer's movement.

```
applyColorOver: aDragContext
"A drag is over the Apply Color button. Test whether a drop would be
    permitted here with this data. If so, return a symbol that indicates the
    feedback to be given to the user. The DragDropManager uses this symbol
    to determine the pointer shape."

    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].
    ^#dropEffectMove
```

8 Add an exit method (applyColorExit:) that accepts a DragDropContext instance as an argument.

9 In the exit method, test the dragged data and return #dropEffectNone if the dragged data is not a color choice.

10 If the dragged data is acceptable, reverse any visual effect that was set in the entry method (in this case, unhighlight the button).

11 Return #dropEffectNone, to signal that no drop has occurred (this method executes only if the pointer leaves the widget without dropping).

```
applyColorExit: aDragContext
"A drag has exited the Apply Color button without dropping. Test whether a
    drop would have been permitted here with this data. If so, restore the
    button to its former state, and return a symbol that indicates the feedback
    to be given to the user."

    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    (self widgetAt: #applyColorButton)
        isInTransition: false.

    ^#dropEffectNone
```

Changing a Button Label During a Drag

Online example: ColorDDExample

This example saves and changes the label of the **Apply More Color** button when the pointer enters the button, restoring the original label when the pointer exits.

1. In the canvas, select the **Apply More Color** button, and set its **ID** property (in this case, enter #applyMoreColorButton).
2. On the **Drop Target** page of a Properties Tool, set the widget's **Entry**, **Over**, and **Exit** properties (enter **applyMoreColorEnter:**, **applyMoreColorOver:**, and **applyMoreColorExit:**).
3. In an entry method (applyMoreColorEnter:), create an IdentityDictionary in which to save the drop target's original state.
4. Save any button characteristics in the IdentityDictionary that are to be restored later. In this case, store the widget and its label. (Storing the widget is a stylistic option that enables the widget to be accessed later through the DragDropContext rather than through the builder.)
5. Get the widget's ConfigurableDropTarget instance from the DragDropContext, and set the IdentityDictionary as its client data.
6. Change the button's label by sending the labelString: message to the button. The message argument is the string to be displayed. Note that a different string is specified depending on the state of the shift key.

```
applyMoreColorEnter: aDragContext
"A drag has entered the bounds of the Apply More Color button. Test whether
    a drop would be permitted here with this data. If so, store the current label
    of the button. Then test whether the shift key is down. Based on this test,
```

```
change the button's label and return a symbol that indicates the feedback to
be given to the user."

  | widget dict |
  aDragContext key == #colorChoice
      ifFalse: [^#dropEffectNone].

  widget := self widgetAt: #applyMoreColorButton .
  dict := IdentityDictionary new.
  dict at: #widget put: widget.
  dict at: #label put: widget label.
  aDragContext dropTarget clientData: dict.

  aDragContext shiftDown
      ifTrue:
          [widget labelString: 'Background'.
          ^#dropEffectCopy].
  widget labelString: 'Foreground'.
  ^#dropEffectMove.
```

7 In an exit method (applyMoreColorExit:), get the drop target's IdentityDictionary from the DragDropContext.

8 Retrieve the button from the IdentityDictionary. (Alternatively, you could obtain the button from the builder.)

9 Retrieve the original label from the IdentityDictionary and put it back on the button. (Note that argument of the label: message is a label object, not a string.)

10 Remove the drop target data from the DragDropContext. This prepares the DragDropContext for the next drop target the pointer may encounter.

```
applyMoreColorExit: aDragContext
"A drag has exited the Apply More Color button without dropping. Test
    whether a drop would have been permitted here with this data. If so, restore
    the button to its former state, and return a symbol that indicates the
```

```
feedback to be given to the user."

    | dict widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    dict := aDragContext dropTarget clientData.
    widget := dict at: #widget.
    widget label: (dict at: #label).
    aDragContext dropTarget clientData: nil.

    ^#dropEffectNone.
```

Tracking a Targeted List Item

Online example: ColorDDExample

This example provides visual feedback for a list in which a drop is intended for a particular item rather than the list as a whole. The steps cause the pointer's location to be indicated by *target emphasis*, a rectangular border around the item containing the pointer. The target emphasis tracks the pointer, scrolling if necessary. (Target emphasis is also used in keyboard traversal of lists to indicate the target for selection.)

1. In the canvas, select the list of color layers and set its **ID** property (in this case, enter #colorLayerList).
2. On the **Drop Target** page of a Properties Tool, set the widget's **Entry**, **Over**, and **Exit** properties (enter colorLayerEnter:, colorLayerOver:, and colorLayerExit:).
3. In an entry method (colorLayerEnter:), create an IdentityDictionary in which to save the drop target's original state.
4. Save any characteristics into the IdentityDictionary that are to be restored later. In this case, store the widget, the location of any target emphasis resulting from keyboard traversal, and a Boolean indicating whether the list has focus.
5. Get the widget's ConfigurableDropTarget instance from the DragDropContext. Store the IdentityDictionary as its client data.
6. Give focus to the list to prepare it for tracking the pointer with target emphasis.

```
colorLayerEnter: aDragContext
"A drag has entered the bounds of the list of color layers. Test whether a drop
    would be permitted here with this data. If so, save the initial state of the
    color layer list, give focus to the list, and return a symbol that indicates the
```

```
feedback to be given to the user."

    | dict widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    widget := self widgetAt: #colorLayerList .
    dict := IdentityDictionary new.
    dict at: #widget put: widget.
    dict at: #targetIndex put: widget targetIndex.
    dict at: #hasFocus put: widget hasFocus.
    aDragContext dropTarget clientData: dict.

    widget hasFocus: true.
    ^#dropEffectMove
```

7 In an over method (colorLayerOver:), retrieve the list widget from the DragDropContext.

8 Send the showDropFeedbackIn:allowScrolling: message to the list to display target emphasis at the pointer's current position, scrolling if necessary. (Remember, this message gets sent each time the pointer moves in the list).

```
colorLayerOver: aDragContext
"A drag is over the list of color layers. Test whether a drop would be permitted
    here with this data. If so, tell the list to scroll the target emphasis when the
    pointer moves. Return a symbol that indicates the feedback to be given to
    the user. The DragDropManager uses this symbol to determine the pointer
    shape."

    | list |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    list := aDragContext dropTarget clientData at: #widget.
    list
        showDropFeedbackIn: aDragContext
        allowScrolling: true.

    ^#dropEffectMove
```

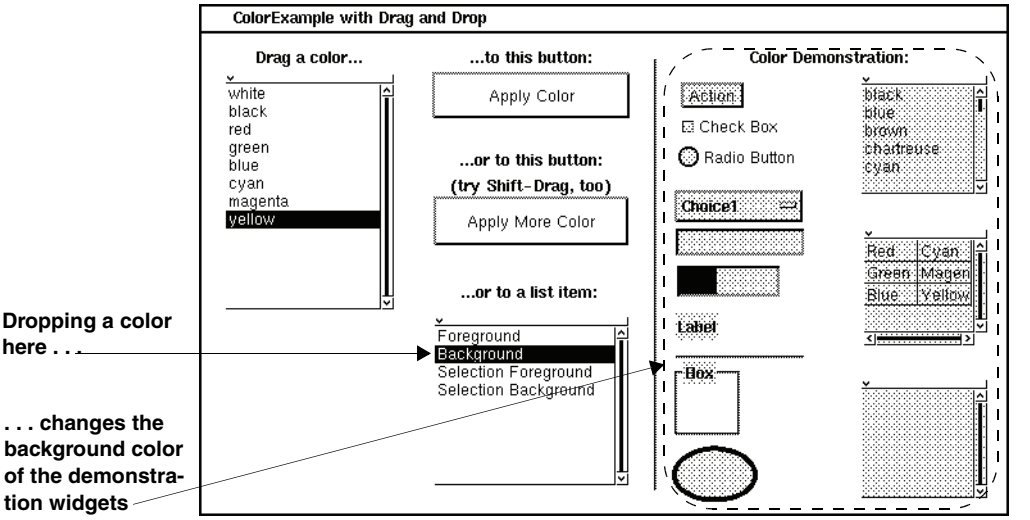
9 In an exit method (colorLayerExit:), get the drop target's IdentityDictionary from the DragDropContext and retrieve the list widget.

10 Restore the list's original target emphasis and focus state.

11 Remove the drop target data from the DragDropContext. This prepares the DragDropContext for the next drop target the pointer may encounter.

```
colorLayerExit: aDragContext
"A drag has exited the list of color layers without dropping. Test whether a
    drop would have been permitted here with this data. If so, restore the initial
    state of the color layer list, and return a symbol that indicates the feedback
    to be given to the user."

    | dict widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].
    dict := aDragContext dropTarget clientData.
    widget := dict at: #widget.
    widget targetIndex: (dict at: #targetIndex).
    widget hasFocus: (dict at: #hasFocus).

    aDragContext dropTarget clientData: nil.
    ^#dropEffectNone
```

Responding to a Drop



A central part of creating a drop target is to implement the action to be taken whenever a drop occurs in it. A typical action is to verify that the dragged data is acceptable to this drop target, and then process that data accordingly. You arrange for this by filling in the widget's **Drop** property on

the **Drop Target** page of the Property Tool. This property specifies the name of the message that the DragDropManager will send when a drop occurs in the widget. You then create a corresponding method in the application model to implement the response.

Like the entry, over, and exit methods, a drop method receives a DragDropContext instance as an argument from the DragDropManager. The method can examine this instance to determine whether to accept the dragged data and, if so, how to process it.

A typical drop method adjusts the appearance of the drop target widget to reverse any visual feedback caused by the enter or over method or to provide visual evidence of the completed drop. The drop method may also need to clean up any drop target data that was created by the entry method.

A drop method returns an effect symbol for the DragDropManager to return to the drag-start method that initiated the drag.

The example sets up the **Apply Color** button in ColorDDExample so that dropping a color on this button applies that color to the foreground layer of the demonstration widgets.

Adding a Drop Response

Online example: ColorDDExample

1. In the canvas, select the widget you want to use as a drop target. In this case, select the **Apply Color** button.
2. On the **Drop Target** page of a Properties Tool, set the widget's **Drop** property (applyColorDrop:). The selector must end with a colon. Apply properties and install the canvas.
3. In a System Browser, add a drop method (applyColorDrop:) in an appropriate protocol (in this case, drop target - button 1). The method must accept a DragDropContext instance as an argument.
4. In the drop method, determine whether the dragged data should be accepted for processing (that is, whether it is a color choice). To do this, send a key message to the DragDropContext instance to obtain the identifying symbol that was assigned when the drag started. If the key is not #colorChoice, return an effect symbol (#dropEffectNone) to signal that no drop is allowed.
5. If the dragged data is acceptable, perform the processing that is to result from the drop. In this example, send a sourceData message to the DragDropContext to obtain the DragDropData; then send this object a

clientData message to obtain the selected color. Turn the selected color into a color value and set it as the foreground color of the demonstration widgets.

6 Restore the widget's original appearance (turn off the highlighting that was turned on by the applyColorEnter: method).

7 Return an effect symbol indicating the result of the drop. This symbol is passed to the drag-start method (colorDrag:), where it can be used to trigger further actions at the drop source.

```
applyColorDrop: aDragContext
"A drop has occurred in the Apply Color button. If the drop is permitted, set
    the foreground color of the demonstration widgets to be the dragged color
    choice. Restore the button to its former visual state and return an effect
    symbol for possible use in the colorDrag method."
    | dict aColor |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].
    dict := aDragContext sourceData clientData.
    aColor := ColorValue perform: (dict at: #colorChoice).
    (self widgetAt: #applyColorButton)
        isInTransition: false.
    self foregroundColor: aColor.
    ^#dropEffectMove.
```

Adding Target Emphasis

Dropping data on a particular list item requires that you enable *target emphasis* in the list through the entry and over methods. Target emphasis tracks the location of the pointer, displaying a rectangular border around the currently targeted list item.

Online example: ColorDDExample

1 In the canvas, select the list of color layers and set its **ID** property (in this case, enter #colorLayerList).

2 On the **Drop Target** page of a Properties Tool, set the widget's **Entry**, **Over** and **Drop** properties (colorLayerEnter:, colorLayerOver:, and colorLayerDrop:).

3 In an entry method (colorLayerEnter:), give focus to the list to prepare it for displaying target emphasis.

```
colorLayerEnter: aDragContext
"A drag has entered the bounds of the list of color layers. Test whether a drop
    would be permitted here with this data. If so, save the initial state of the
    color layer list, give focus to the list, and return a symbol that indicates the
```

```
    feedback to be given to the user."

    | dict widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    widget := self widgetAt: #colorLayerList .
    dict := IdentityDictionary new.
    dict at: #widget put: widget.
    dict at: #targetIndex put: widget targetIndex.
    dict at: #hasFocus put: widget hasFocus.
    aDragContext dropTarget clientData: dict.

    widget hasFocus: true.
    ^#dropEffectMove
```

4 In an over method (colorLayerOver:), retrieve the list widget from the DragDropContext and send it the showDropFeedbackIn:allowScrolling: message to display target emphasis at the pointer's current position.

```
colorLayerOver: aDragContext
"A drag is over the list of color layers. Test whether a drop would be permitted
    here with this data. If so, tell the list to scroll the target emphasis when the
    pointer moves. Return a symbol that indicates the feedback to be given to
    the user. The DragDropManager uses this symbol to determine the pointer
    shape."

    | list |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    list := aDragContext dropTarget clientData at: #widget.
    list
        showDropFeedbackIn: aDragContext
        allowScrolling: true.

    ^#dropEffectMove
```

5 In a drop method (colorLayerDrop:), test whether the dragged data is a color choice; if so, obtain the selected color from the dragged data and turn it into a color value.

6 Send a targetIndex message to the list to get the index of the targeted list item (the item containing the pointer when the drop occurs).

7 Get the color layer that is shown in the list at the targeted index.

8 Give visual feedback to indicate a successful drop. In this case, cause the targeted list item to appear selected (set the list's selection

index to be the targeted index). Alternatively, you could restore the list to its original visual state, as is done in the colorLayerExit: method.

9 Use the targeted color layer to choose the appropriate message for changing the color of the demonstration widgets.

```
colorLayerDrop: aDragContext
"A drop has occur in the list of color layers. If the drop is permitted, combine
    the dragged color choice and the targeted color layer to change the color of
    the appropriate parts of the demonstration widgets. Return an effect
    symbol for possible use in the colorDrag method."

    | dict aColor widget idx aLayer |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    dict := aDragContext sourceData clientData.
    aColor := ColorValue perform: (dict at: #colorChoice).

    widget := aDragContext dropTarget clientData at: #widget.
    idx := widget targetIndex.
    idx = 0 ifTrue: [^#dropEffectNone].
    aLayer := self colorLayer listHolder value at: idx.

    self colorLayer selectionIndexHolder value: idx.

    aDragContext dropTarget clientData: nil.

    aLayer = 'Foreground'
        ifTrue: [self foregroundColor: aColor].
    aLayer = 'Background'
        ifTrue: [self backgroundColor: aColor].
    aLayer = 'Selection Foreground'
        ifTrue: [self selectionForegroundColor: aColor].
    aLayer = 'Selection Background'
        ifTrue: [self selectionBackgroundColor: aColor].

    ^#dropEffectMove.
```

Examining the Drag Context

Drop target methods can examine a variety of information by querying the DragDropContext instance that is passed to them by the DragDropManager. Among other things, the DragDropContext instance contains the DragDropData you created in the **Drag Start** method, plus the current pointer location and modifier key states. This information can be useful for determining the appropriate drop response.

In a drop target method, obtain information about the drag from the DragDropContext instance that is passed to the method (assume the argument name is aDragContext).

```
"Get the key that was assigned to the dragged data."

    aDragContext key.
"Get the application model from which the drag originated."

    aDragContext sourceData contextApplication.
"Get the window from which the drag originated."

    aDragContext sourceData contextWindow.
"Get the current pointer location."

    aDragContext mousePoint.
```

Responding to Modifier Keys

You can make drag and drop sensitive to the state of the <Control>, <Shift>, <Alt>, and <Meta> modifier keys. For example, in many applications, a user can move a file by dragging it and copy a file by <Shift>-dragging it.

Making drag and drop sensitive to modifier keys involves:

- Providing the appropriate visual feedback in the drop target's entry, over, and exit methods.
- Providing appropriate processing in the drop target's drop method.

The example sets up the **Apply More Color** button in ColorDDExample so that dragging changes the foreground color of the demonstration widgets, while <Shift>-dragging changes the background color.

Responding to a Modified Drag

Online example: ColorDDExample

1. In the canvas, select the list of color layers and set its **ID** property (in this case, enter #applyMoreColorButton).
2. On the **Drop Target** page of a Properties Tool, set the widget's **Entry**, **Over**, **Exit** and **Drop** properties (applyMoreColorEnter:, applyMoreColorOver:, applyMoreColorExit:, and applyMoreColorDrop:). Apply properties and install the canvas.
3. In an entry method (applyMoreColorEnter:), send a shiftDown message to the DragDropContext instance to find out whether the user is pressing the <Shift> key down.
4. If the <Shift> key is down, provide appropriate visual feedback. In this case, change the button's label to indicate that background colors will be set, and return an effect symbol (#dropEffectCopy) to signal a modified transfer.
5. If the <Shift> key is not down, change the button's label to indicate that foreground colors will be set, and return an effect symbol (#dropEffectMove) to signal a regular transfer.

```
applyMoreColorEnter: aDragContext
"A drag has entered the bounds of the Apply More Color button. Test whether
    a drop would be permitted here with this data. If so, store the current label
    of the button. Then test whether the shift key is down. Based on this test,
    change the button's label and return a symbol that indicates the feedback to
    be given to the user."

    | widget dict |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    widget := self widgetAt: #applyMoreColorButton .
    dict := IdentityDictionary new.
    dict at: #widget put: widget.
    dict at: #label put: widget label.
    aDragContext dropTarget clientData: dict.

    aDragContext shiftDown
        ifTrue:
            [widget labelString: 'Background'.
            ^#dropEffectCopy].
    widget labelString: 'Foreground'.
    ^#dropEffectMove.
```

6 In an over method (applyMoreColorOver:), find out whether the user has changed the <Shift> key state while dragging the pointer within the widget. (Remember, this method executes each time the pointer moves in the widget.) If the <Shift> key is down, test whether the button's label needs to change; if so, change it. Return the #dropEffectCopy symbol.

7 If the <Shift> key is not down, test whether the button's label needs to change; if so, change it. Return the #dropEffectMove symbol.

```
applyMoreColorOver: aDragContext
"A drag is over the Apply More Color button. Test whether a drop would be
    permitted here with this data. If so, test whether the shift key is down.
    Based on this test, return a symbol that indicates the feedback to be given
    to the user. The DragDropManager uses this symbol to determine the
    pointer shape."

    | widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    widget := aDragContext dropTarget clientData at: #widget.

    aDragContext shiftDown
        ifTrue:
            [widget label text string = 'Background'
                ifFalse: [widget labelString: 'Background'].
            ^#dropEffectCopy].

    widget label text string = 'Foreground'
        ifFalse: [widget labelString: 'Foreground'].
    ^#dropEffectMove.
```

8 In an exit method (applyMoreColorExit:), restore the button's original label. In this example, the same label is restored, regardless of the <Shift> key's state.

```
applyMoreColorExit: aDragContext
"A drag has exited the Apply More Color button without dropping. Test
    whether a drop would have been permitted here with this data. If so, restore
    the button to its former state, and return a symbol that indicates the
```

```
feedback to be given to the user."

    | dict widget |
    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    dict := aDragContext dropTarget clientData.
    widget := dict at: #widget.
    widget label: (dict at: #label).
    aDragContext dropTarget clientData: nil.

    ^#dropEffectNone.
```

9 In a drop method (applyMoreColorDrop:), test whether the <Shift> key is down.

10 If the <Shift> key is down, apply the dragged color choice to the background color layer of the demonstration widgets. Return #dropEffectCopy to signal a modified transfer.

11 If the <Shift> key is not down, apply the dragged color choice to the foreground color layer. Return #dropEffectMove to signal a regular transfer. Note that a drag-start method could respond differently depending on which symbol is returned.

```
applyMoreColorDrop: aDragContext
"A drop has occured in the Apply More Color button. If the drop is permitted,
obtain the dragged color. Then test whether the shift key is down. If so, set the
background color of the demonstration widgets. If not, set their foreground
color. Restore the button to its former visual state and return an effect symbol
for possible use in the colorDrag method."

    | dict widget aColor |

    aDragContext key == #colorChoice
        ifFalse: [^#dropEffectNone].

    dict := aDragContext sourceData clientData.
    aColor := ColorValue perform: (dict at: #colorChoice).

    dict := aDragContext dropTarget clientData.
    widget := dict at: #widget.
    widget label: (dict at: #label).
    aDragContext dropTarget clientData: nil.

    aDragContext shiftDown
        ifTrue:
            [self backgroundColor: aColor.
            ^#dropEffectCopy].
    self foregroundColor: aColor.
    ^#dropEffectMove
```

7

Dialogs

Dialogs are special purpose windows, typically used to display notices or to prompt for specific information required by an application. VisualWorks provides a simple method for displaying many forms of common dialogs, as well as a full capability for creating custom dialogs.

A variety of dialogs can be constucted quite simply by sending various class methods to the Dialog class. For more complex and custom dialogs, SimpleDialog provides a variety of utility methods to simplify dialog creation.

Standard Dialogs

The standard dialogs provided in VisualWorks make it simple to display commonly used dialogs from within an application. Typically, all you need to do is send a message to the Dialog class that identifies the kind of dialog and any specific information required, such as text.

The standard dialogs are demonstrated in the example DialogExample.

Displaying a Warning

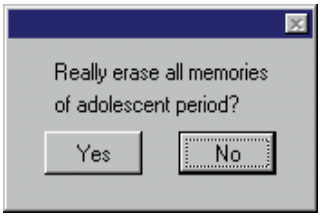
A warning dialog is frequently used when an action cannot be completed, such as if a search command cannot find a user-specified string.

A warning dialog generally displays a simple textual message, which can have embedded carriage returns and text styles.

A warning dialog is displayed by sending a warn: message to the Dialog class. When the user clicks **OK**, the message returns the value nil.

```
warn
    | returnVal |
    returnVal := Dialog
        warn: 'The memory named\''FirstKiss''\was not found.\'
        withCRs.

    self returnedValue value: returnVal printString.
```

Asking a Yes/No Question



Frequently an application needs to ask the user a yes/no question, such as to request verification when the user has initiated an action that may have unintended side effects. A yes/no dialog is often referred to as a *confirmer*.

By convention, the question is phrased in such a way that a **Yes** answer causes the action to proceed.

A confirmer dialog is displayed by sending a confirm: message to the Dialog class, with a string containing the question.

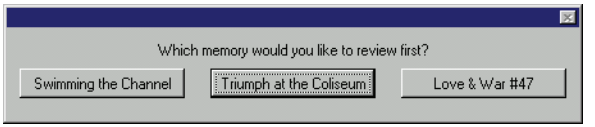
```
Dialog confirm: 'Really erase all memories\of adolescent period?\'
    withCRs.
```

If the user clicks **Yes**, the dialog returns true, and if the user clicks **No**, the dialog returns false.

The default answer is **Yes**. In hazardous situations, this might not be a good default. To change the default response, send a confirm:initialAnswer: message to Dialog. The second argument is either true or false.

```
Dialog
    confirm: 'Really erase all memories\of adolescent period?'
        withCRs
        initialAnswer: false
```

Multiple-Choice Dialog



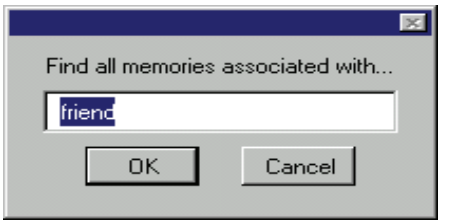
Frequently more choices than Yes/No, or other choices are required, but still only a small number. The multiple-choice dialog provides this capability. The choices are arrayed as a horizontal row of buttons.

The message that creates a multiple-choice dialog assigns a symbol to each choice. When the user clicks a choice, the message returns the corresponding symbol to the application.

To display a multiple-choice dialog, send a choose:labels:values:default: message to the Dialog class. The choose argument is the question. The labels argument is an array of strings to be displayed on the answer buttons. The values argument is an array of Symbols to be used as return values by the answer buttons. The default argument is the Symbol that is associated with the desired default answer.

```
Dialog
    choose: 'Which memory would you like to review first?'
    labels: #( 'Swimming the Channel'
        'Triumph at the Coliseum'
        'Love & War #47')
    values: #(#swim #triumph #love47)
    default: #triumph
```

Requesting a Textual Response



To prompt for a short, textual response, use the request dialog, which displays a fill-in-the-blank input field and a label.

To open a request dialog, send request: to the Dialog class, with text describing what is being requested:

```
Dialog request: 'Find all memories associated with...'
```

When the user enters a string and clicks **OK**, the dialog returns the user-specified string. If the user clicks **Cancel**, the default is to return an empty string.

By default, an empty string appears in the input field. To specify a default response, send request:initialAnswer: to Dialog, with the default answer as a string:

```
Dialog
    request: 'Find all memories associated with...'
    initialAnswer: 'friend'
```

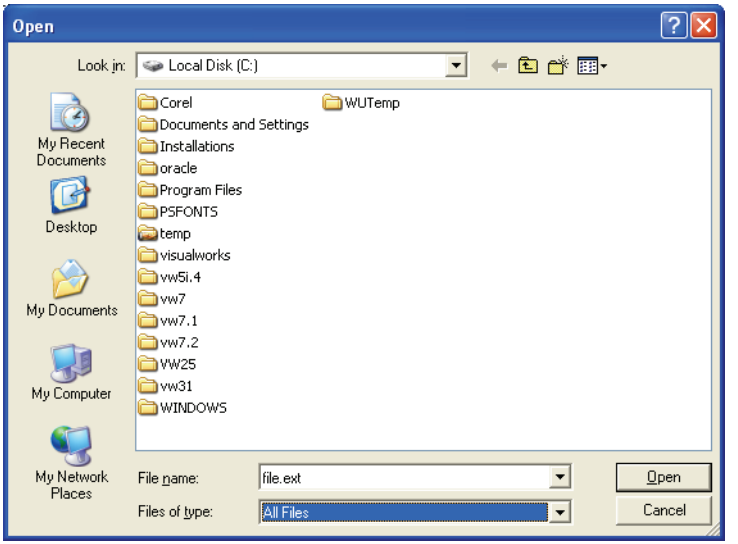
If an empty string is not an appropriate response to **Cancel**, you can specify an alternate action or value. Send request:initialAnswer:onCancel: to Dialog, with a block containing the action to be taken or the value to be returned:

```
getText
    | returnVal |
    returnVal := Dialog
        request: 'Find all memories associated with...'
        initialAnswer: 'friend'
        onCancel: [self defaultRuminationTopic].

    "Update the text field in the main window."
    self returnedValue value: returnVal printString.
```

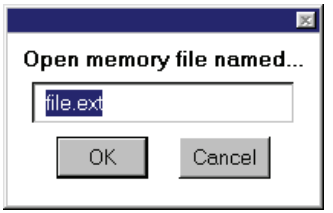
Requesting a Filename

The file dialog varies with the operating system.

On Windows platforms, a native Windows file browser dialog is used, allowing the familiar operations.



On other systems, the filename dialog is a special case for a fill-in-the-blank dialog that prompts the user for a pattern and displays a list of file names matching that pattern. The dialog accepts * and # as wildcard characters.



When the user clicks **Cancel**, an empty string is returned by default. The final variant shows how to arrange for a different value to be returned, an action to be taken, or both.

To open a file request dialog, send requestFileName: to the Dialog class, with a string for the label (the argument is ignored by the Windows file dialog):

```
Dialog requestFileName: 'Open memory file named...'
```

In some situations, there may be a natural default file name, as when saving a file that has been edited. To provide a default filename, send requestFileName:default: to Dialog, with a string containing the file name:

```
Dialog
    requestFileName: 'Open memory file named...'
    default: 'hero01.mem'
```

Note that on Windows platforms, while navigating the file system in the dialog, the underlying OS current directory changes following the currently displayed directory. This can cause problems in multi-threaded applications (multi-proc UI) if file access operations that rely on a relative path are being performed in one process while the file dialog is running in another process. In this case, the file operation will follow the current directory as determined by the file dialog, giving the wrong results in the file accessing process. The original directory is restored as current once the dialog closes.

Accordingly, in multi-process Windows applications, and multi-threaded applications in general, file access operations should not rely on a relative path, but convert it to an absolute path first, and use that for the access path.

Handling a File that Exists or Does Not Exist

By default, the dialog accepts any filename that is accepted by the operating system. In some circumstances, confirmation may be required if the specified file either already exists or does not exist.

If you expect the file to exist, you will want confirmation if it does not. In this case, send requestFileName:default:version: to Dialog, with the symbol #new as argument:

```
Dialog
        requestFileName: 'Open memory file named...'
        default: 'hero01.mem'
        version: #new
```

On the other hand, if you expect the file to be new, you want confirmation if the file already exists. In this case, send requestFileName:default:version: with the symbol #old.

In other cases, it may be preferable to cancel the operation if the file either exists or does not exist. To cancel if the file does exist, send requestFileName:default:version: with the symbol #mustBeNew.

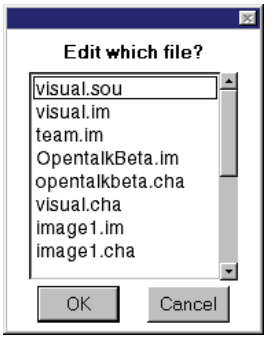
```
Dialog
        requestFileName: 'Open memory file named...'
        default: 'hero01.mem'
        version: #mustBeNew
```

To cancel if the file does not exist, send requestFileName:default:version: with the symbol #mustBeOld.

Further cancellation processing can be specified by sending a requestFileName:default:version:ifFail: message to Dialog. The final argument is a block containing the action to be taken or the value to be returned:

```
getFilename
    | returnVal |
    returnVal := Dialog
        requestFileName: 'Open memory file named...'
        default: 'hero01.mem'
        version: #mustBeOld
        ifFail: [Transcript show: 'Memory file access canceled'. ''].
    "Update the text field in the main window."
    self returnedValue value: returnVal printString.
```

Multiple-Selection List Dialog



A multiple-selection list dialog displays a list of selectable items. This dialog is similar to a multiple-selection list widget, but is displayed only when needed, separately from the main window. Each item in the list is associated with a value, like a menu item, and your application can either insert the selected value in a value holder or trigger an action.

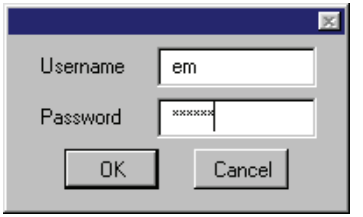
By default, the dialog contains a list of items, an **OK** button, and a **Cancel** button, but additional buttons can be added.

To open a multiple-selection dialog, send choose:fromList:values:lines:cancel: to the Dialog class. The choose: argument is a prompt string. The fromList: argument is a collection of strings, the list item labels. The values: argument is a collection of the same size as the fromList: collection, containing the values to be associated with the list items. The lines: argument is an integer indicating the maximum number of list items to display (for a long list). The cancel: argument is a block containing the action to take when the **Cancel** button is pressed.

```
| files response |
files := Filename defaultDirectory directoryContents
        reject: [ :name | name asFilename isDirectory].

response := Dialog
        choose: 'Edit which file?'
        fromList: files
        values: files
        lines: 8
        cancel: [^nil].

response asFilename edit.
```

Prompting for a Password

The following is an example of using SimpleDialog interface construction methods to assemble and format this dialog. Notice the use of the #password type, which masks the password when entered as a string of asterisks.




```
| user pass |
(SimpleDialog initializedFor: nil)
        setInitialGap;
        addMessage: 'Username' textLine: (user := '' asValue) boundary: 0.4;
        addGap;
        addMessage: 'Password' textLine: (pass := '' asValue)
            type: #password boundary: 0.4;
        addGap;
        addOK: [true];
        openDialog.
user value -> pass value
```

Supplying Extra Action Buttons Below the List

If the two default buttons are not enough or not the correct buttons for your application, you can change them. For example, when you enter a wildcard pattern in a file pathname dialog, a list dialog shows the files that match your pattern and offers a **Try again** button in case you want to try a different pattern.

To configure buttons, send choose:fromList:values:buttons:values:lines:cancel: to the Dialog class. The buttons: argument is a collection of strings to be used as button labels. The values: argument is a collection of values to be associated with the button labels.

```
| files response |
files := Filename defaultDirectory directoryContents
        reject: [ :name | name asFilename isDirectory].

response := Dialog
    choose: 'Edit which file?'
    fromList: files
    values: files
    buttons: #('Count Files')
    values: #(#count)
    lines: 12
    cancel: [^nil].

response == #count
    ifTrue: [Dialog warn: files size printString]
    ifFalse: [response asFilename edit]
```

Linking a Dialog to a Master Window

By default, the built-in dialogs use system settings for their colors and UI Look. If your application employs a special set of settings, you can arrange for dialogs to mimic the colors and UI Look of a master window.

In addition, some window systems create a visual connection between a dialog and its master window.

Adopting the Look of a Master Window

To integrate well with an application, it is often necessary for dialogs to adopt the look of its master window. The standard dialogs have variant opening message forms ending with a for: keyword. The argument is a window, typically the current window, which is given as shown in the example:

```
confirm
    | returnVal |
    returnVal := Dialog
        confirm: 'Really erase all memories\of adolescent period?'
        withCRs
        initialAnswer: false
        for: Window activeController view.

    "Update the text field in the main window."
    self returnedValue value: returnVal printString.
```

Linking a Dialog to a Window

You can link a dialog to the currently active window with a warn:for: message. The master window is typically the main application window, which an application model can access through self mainWindow.

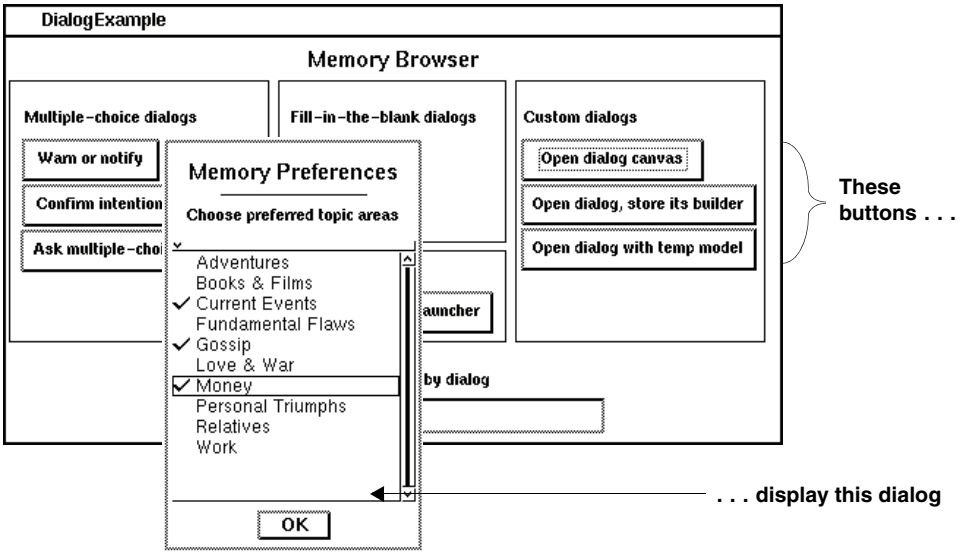
To link a dialog to a window, send a useColorOveridesFromParent: message to the SimpleDialog class. The argument true causes subsequently opened dialogs to adopt the colors of their master window, in addition to the UI look. (Note that “Overides” is misspelled in the method name and must therefore be misspelled here.)

Then send the for: keyword form of the dialog opening message to the Dialog class, with the master window as the argument for the for: keyword.

```
| masterWindow |
SimpleDialog useColorOveridesFromParent: true.
masterWindow := ScheduledWindow new.
masterWindow background: ColorValue yellow.
masterWindow open.

Dialog
    warn: 'This dialog has a yellow background, too.'
    for: masterWindow.

masterWindow sensor eventQuit: nil.
```

To reset the SimpleDialog class to its default behavior, send the useColorOveridesFromParent: message with the argument false.

Creating a Custom Dialog



When a standard dialog is not sufficient, you can create your own using the canvas tool. You can then open the interface specification in a dialog window.

The basic technique is send an openDialogInterface: message to the application model, with the symbolic name of the dialog's interface specification.

```
openDialogCanvas
    | returnVal |
    returnVal := self openDialogInterface: #memoryZonesDialog.
    "Update the text field in the main window."
    self returnedValue value: returnVal printString.
```

The dialog is created as an instance of SimpleDialog, which provides its own interface builder for setting up the dialog's widgets. The dialog builder obtains any needed value models, actions, and resources for its widgets from the application model.

Note, however, that buttons whose **Action** properties are #accept or #cancel obtain their actions from the SimpleDialog instance instead. These predefined actions are useful for **OK** and **Cancel** buttons on the dialog. Consequently, if you define methods named accept or cancel in the application model, they will be ignored.

An alternative technique for creating a custom dialog (not illustrated here) is to create a separate model for the dialog (typically, a subclass of SimpleDialog). You install the dialog's canvas in this subclass and then program the subclass to provide the value models, actions, and resources needed by the dialog's widgets. This technique enables you to reuse the dialog more easily in further applications.

Providing a Temporary Model for the Dialog

You can create a dynamic dialog by programming the application model to create an instance of SimpleDialog and configure its interface builder. This creates a temporary model for the dialog, which is useful when the value models for the dialog's widgets are only needed during the lifetime of the dialog. For example, a file-finding dialog might employ several widgets, each requiring a value model, but only the ultimate filename is of interest to the application.

In the example, the properties for the dialog's list widget tell the dialog's builder that the list widget needs a MultiSelectionInList for its value holders.

1. In the method that is to open the dialog, create an instance of SimpleDialog.
2. Get the builder from the SimpleModel and preload it with one binding for each active widget. The aspectAt: argument is the symbol you specified in the widget's **Aspect** property. The put: argument is an appropriate value model.
3. Ask the SimpleDialog to open the interface.

```
openTempModelDialog
    | returnVal dialogModel list |
    dialogModel := SimpleDialog new.
    dialogBuilder := dialogModel builder.

    "Since the simple model does not respond to a #memoryZones message,
    its builder must be preloaded with a multilist."
    list := MultiSelectionInList new
        list: self memoryZones list copy.
    dialogBuilder aspectAt: #memoryZones put: list.

    "Open the interface."
    returnVal := dialogModel
        openFor: self
        interface: #memoryZonesDialog.

    "Update the text field in the main window."
    self returnedValue value: returnVal printString.
```

8

Custom Views

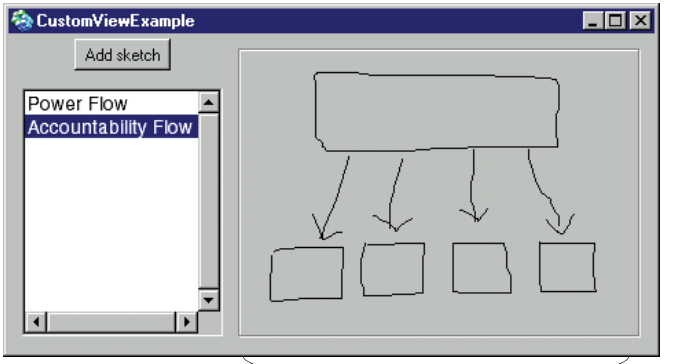
A view displays text or graphics representing all or part of a data model. Each of the widgets uses a view to display a data model. When an existing widget does not serve your purpose, you can create a custom view.

A custom view is like a small application. It typically has a domain model, a controller, and a view, which is similar to an application model. You begin that process by creating the view class, and then add its UI specification and connect it to its domain model.

Building a custom view is described in the context of an example, CustomViewExample, a simple sketch pad. It uses four classes that demonstrate the interactions among a domain model (Sketch), a custom view (SketchView), a custom controller (SketchController), and an application model (CustomViewExample). SketchView uses a Sketch as its model, which has a name and a collection of points representing a series of sketched strokes. SketchView uses a SketchController to handle mouse and keyboard input.

These examples classes are all in the CustomView-Example parcel.

Creating a View Class



custom view for sketching

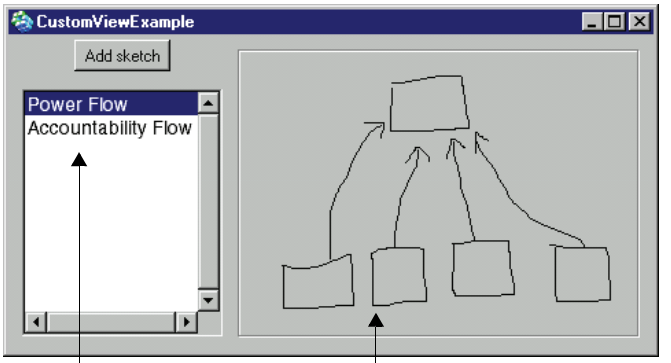
The new view requires a class, which you create as a subclass of View or one of its subclasses.

Online example: SketchView

1. In a System Browser, select **Class → New Class...** to display the class-definition dialog.
2. In the dialog, specify the name space, class name, and superclass name. The superclass is typically either View (as in the example, or a subclass of View). You can also supply any instance variable names.
3. Click **Accept** to create the definition.

```
Smalltalk.Examples defineClass: #SketchView
    superclass: #{UI.View}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'Examples-Help'
```

Connecting a View to a Domain Model



A view displays text or graphics that communicate the state of its domain model, or at least a portion of its domain model. Since a view must communicate frequently with the domain model, it needs a way of accessing that object. As a subclass of DependentPart, every view inherits an instance variable for storing its model. Sending a model: message to the view, typically when the view is created, stores the model in this instance variable, where it can be accessed easily.

A side effect of the model: message is that the view is registered as a dependent of the model. This link sets the stage for the view to update its display when the model changes.

About the example: Although some views have the same domain model for their whole lifetimes, SketchView changes its model each time the user selects a different Sketch. For that reason, SketchView reimplements the model: method so it can update its display after storing the new model.

Online example: CustomViewExample, SketchView

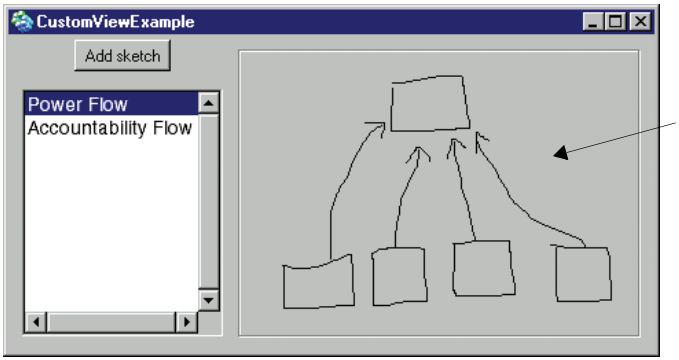
1 Tell the view which object to use as its domain model. This is done in an initialization method or, as in the example, the application model (CustomViewExample) can notify the view whenever the domain model changes.

```
changedSketch
    self sketchView model: self sketches selection.
```

2 If the view needs to take action when its model is changed, such as redisplaying itself, override the inherited model: method (as in SketchView).

```
model: aModel
    super model: aModel.
    self invalidate.

    "Tell the controller where to send menu messages."
    self controller performer: aModel.
```

Defining What a View Displays



The view gets a collection of points from the model and displays the points as line segments

A view's purpose is to display text or graphics. It does so in a method named displayOn:, which is sent to the view whenever circumstances require that it update its display.

The view decides what to display based on the state of its domain model.

It displays the text and/or graphics on a GraphicsContext, which is an object that windows and other display surfaces use for rendering objects.

Online example: SketchView

In CustomViewExample, a SketchView is used to display the line segments that are stored in its domain model, a Sketch.

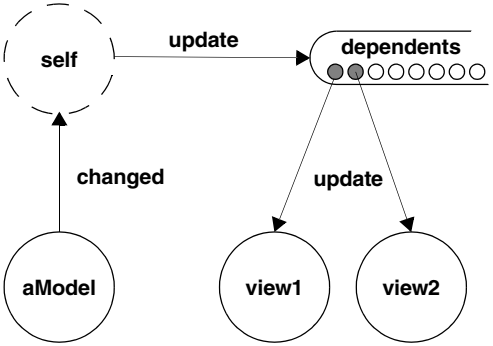
1 In a displaying protocol, add a displayOn: method to the view. The argument is a GraphicsContext.

2 In the displayOn: method, get the required data from the model (in the example, a set of line segments, each represented as a collection of points).

3 In the displayOn: method, display the appropriate text or graphics, based on the data from step 2 (in the example, each collection of points is displayed as a Polyline).

```
displayOn: aGraphicsContext
    self model isNil ifTrue: [^self].

    self model strokes do: [ :stroke |
        aGraphicsContext displayPolyline: stroke].
```

Updating a View When Its Model Changes



Since the purpose of a view is to display some aspect of its domain model, it must be prepared to change its display when the model is changed.

When the domain model changes its state, it is responsible for notifying all of its dependents. It does so by sending a variant of the changed:with: message to itself. The first argument is a Symbol indicating what was changed, and the second argument is the new value.

Shorter versions of the changed:with: message, changed: and changed, can be used when less information needs to be sent. Read their comments in a code browser for more information.

The changed:with: message is inherited, and it sends an update:with: message to each dependent, passing along the same two arguments. Thus, the view must implement an update:with: method in which it gets the new data from the model and displays it.

Online example: Sketch, SketchView

1 In any method in the domain model that changes the model in a way that affects the view, send a variant of the changed:with: message to the model. (In the example, Sketch sends three such messages, one

when it adds a point and the others when it erases some or all of its contents.)

```
add: aPoint
    "Add aPoint to the current stroke."

    self strokes last add: aPoint.
    self changed: #stroke with: self currentLineSegment.

eraseLine
    "Erase the last stroke that was drawn."

    self strokes isEmpty
        ifFalse: [
            self strokes removeLast.
            self changed: #erase with: nil].

eraseAll
    "Erase my contents."

    self strokes removeAll: self strokes copy.
    self changed: #erase with: nil.
```

2 In the view, implement a variant of the update:with: method to take the appropriate action in response to a change in the model. (In the example, the same update:with: method responds to either of the changed:with: messages sent by the model.)

```
update: anAspect with: anObject
    "When a point is added to the model..."
    anAspect == #stroke
        ifTrue: [anObject asStroker displayOn: self graphicsContext].

    "When the model erases its contents..."
    anAspect == #erase
        ifTrue: [self invalidate].
```

Connecting a View to a Controller

If a view responds to mouse or keyboard input, it needs a controller to process mouse and keyboard input. Such a view is called *active*. A view is often closely allied with its controller, so an inherited mechanism installs the desired controller when the view is created.

Controllers are event-driven in VisualWorks, meaning that they capture mouse and keyboard events passed into VisualWorks from the operating system. An view's controller identifies which events it responds to, and how it responds.

SketchView uses a SketchController, which changes the cursor to a crosshair, notifies the model when the user draws with the <Select> button, and provides a menu when the <Operate> button is pressed.

Creating a Controller Class

Because an input controller defines the interactive character of a view, changing the controller can have a dramatic impact on the operation of a view. When an existing controller class does not serve your purposes, you can create a custom controller class.

An event-driven controller that responds to keyboard input needs to have an instance variable named keyboardProcessor, and accessor methods for that variable (keyboardProcessor and keyboardProcessor:). The controller should not initialize the variable, however, because a KeyboardProcessor will be supplied by the window using a keyboardProcessor: message.

By default, an event-driven controller does not accept keyboard focus. When it handles keyboard input, it must respond true to a desiresFocus message.

To create the basic controller class:

1 In a System Browser, define a new controller class.

In the template, specify the name space, the class name, and the superclass. The superclass is some class in the Controller hierarchy. For this example it is ControllerWithMenu.

2 Supply variable names, if any, and then **Accept** the definition.

In our example, a variable named strokeInProgress is created to store a true/false indication of whether the user is actively sketching, and a keyboardProcessor variable to specify whether it accepts keyboard input

SketchController's definition is:

```
Smalltalk.Examples defineClass: #SketchController
    superclass: #{UI.ControllerWithMenu}
    indexedType: #none
    private: false
    instanceVariableNames: 'strokeInProgress keyboardProcessor '
    classInstanceVariableNames: ''
    imports: ''
    category: 'Examples-Help'
```

3 Add an initialize method to assign an initial value to the instance variables. Remember that we do not initialize keyboardProcessor.

```
initialize

    super initialize.
    strokeInProgress := false.
```

4 Add accessor methods for the instance variables. The methods are responsible for getting and setting the value of each instance variable.

```
strokeInProgress
    ^strokeInProgress

strokeInProgress: aBoolean
    strokeInProgress := aBoolean

keyboardProcessor
    ^keyboardProcessor

keyboardProcessor: kp
    keyboardProcessor := kp
```

5 When keyboard input is to be handled, add a desiresFocus message to the controller. The method simply returns true, overriding the inherited method, which returns false.

```
desiresFocus
    ^true
```

Connect the Controller to the Model

A controller's purpose is to respond to input events. Frequently, the response involves sending a message to the view's domain model.

A controller inherits a model instance variable (from Controller) for storing the model so it can easily access that object. Also, by default, a view sends a model: message to its controller when it receives a model: message itself, effectively installing the model in both the view and its controller.

It the default behavior is not appropriate, you can also set the controller's model explicitly. This is not necessary for our example. If it is necessary for your application, simply send a model: message to the controller. The argument is the model.

Connect the Controller to the View

Online example: SketchView, SketchController

An application model sometimes needs to access a view's controller. The usual way of accessing the controller is to ask the view for it. Thus, the view must itself be able to access its controller.

A view inherits an instance variable for storing its controller. By default, this instance variable is initialized automatically, when a view is opened, according to the value returned by its defaultControllerClass method.

```
defaultControllerClass
    ^SketchController
```

To make a view *passive*, return NoController from the defaultControllerClass method.

Alternatively, you can install a controller in a view directly, by sending a controller: message to the view, with the controller class name as argument.

Connecting a Composite View to a Controller

When multiple views inhabit the same window, normally they have separate controllers. In some situations, the composite object that groups them needs its own controller, either instead of the individual controllers or in addition to them. A CompositeView is intended for grouping views that need a common controller. To initialize its controller:

Send a controller: message to the composite that groups the views. The argument is an instance of the desired type of controller (not the class name as with defaultControllerClass).

Redisplaying All or Part of a View

A view can redisplay its entire contents or just a portion of them. For example, when one window overlaps another, the overlap region is all that needs to be redisplayed when the lower window is no longer obscured by the upper window. This overlap region is called a *damage rectangle*, because it is a rectangular region that was damaged by an overlapping window.

The window's sensor keeps track of such damage rectangles and repairs them in a batch to avoid repairing the same region twice. Sending invalidate to a view causes the entire view to be treated as a damage rectangle. Alternatively, you can limit the damage rectangle to a portion of the window.

By default, damage rectangles are accumulated until the window's controller reaches a certain point in its cycle of activity. That is sufficient in most situations. However, when a competing process is monopolizing the processor, the delay can be significant. In this case, you can force the damage to be repaired immediately.

Invalidating a view is done in a view method, when the view updates its model. It can also be done by an application model that has changed a widget's data model in a way that bypasses the normal dependencies.

Redisplaying a View

Online example: SketchView

Send invalidate to the view. This is typically done in a view method that changes the model (as in the example).

```
model: aModel
    super model: aModel.
    self invalidate.

    "Tell the controller where to send menu messages."
    self controller performer: aModel.
```

Redisplaying Part of a View

Send invalidateRectangle: to the view. The argument is a rectangle that represents all or part of the view's bounding box. The bounding box can be accessed by sending bounds to the view.

Redisplaying Immediately

Send invalidateRectangle:repairNow: to the view. The first argument is a rectangle that represents all or part of a view's bounding box. The second argument is true when immediate redisplay is desired, and false for the default behavior.

Integrating a View into an Interface

A View Holder widget is provided on the UI Painter Palette for integrating a custom view into a canvas. This view holder enables you to treat your custom view like a standard widget in that you can paint its layout and apply borders and scroll bars. However, your application is responsible for connecting the view to a domain model.

Online example: CustomViewExample, SketchView

1. Select a View Holder from the Palette and place it on the canvas.
2. In the view holder's **View** property, enter the name of the application-model method that supplies an instance of the desired view (sketchView).
3. If the application model will need to access the custom view while the application is running, use a System Browser to create an instance variable (sketchView) in which to store the custom view.
4. Use a System Browser to create the application-model method that you named in step 2 (sketchView). This method typically answers the contents of the instance variable that you created in step 3.

```
sketchView
    ^sketchView
```

5. In an initialize method in the application model, create an instance of the custom view. If appropriate, connect the custom view to a data model. (In the example, there is no model to be connected until the user adds the first Sketch object.)

```
initialize
    sketches := SelectionInList with: OrderedCollection new.
    sketches selectionIndexHolder
        onChangeSend: #changedSketch to: self.

    sketchView := SketchView new.
```

9

Configuring Widgets

VisualWorks provides a rich set of widgets for building the GUI for your application. The widgets are shown in the canvas's palette and can be selected and added to design a window, as described in Chapter 2, "Building an Application's GUI."

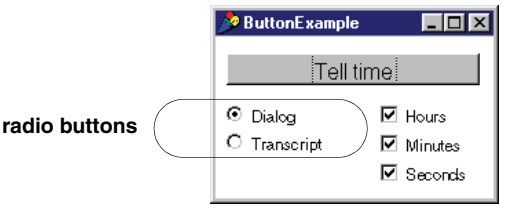
This chapter describes the widgets and how to connect them to your application.

Buttons

Online example: ButtonExample

Radio button, checkboxes, and action buttons are common GUI elements for setting standard parameters and starting actions. Radio buttons and checkboxes are used to select one or more options. Action buttons frequently begin an operation or invoke another window.

Radio Buttons



A group of radio buttons allows a user to select a single item from a limited list of choices. Selecting a radio button causes any other button in its group to be deselected. This characteristic makes radio buttons useful only where an exclusive selection is appropriate.

Radio buttons are typically used only for a very brief and static set of choices. If you want your application to reconfigure the list of choices programmatically, you use a list widget instead. A list is also scrollable, making it more suitable for a long list of options.

To allow users to select more than one choice, use either a group of check boxes or a list widget with the multi-select property turned on.

To add a group of radio buttons:

1. Add a radio button to the canvas for each selectable option.
2. For each button, change the **Label** property to name the choice.
3. Enter the *same* **Aspect** property for each button in the group.

   The common aspect is what makes a group.
4. For each button, enter a *different* **Select** property.

   This symbol is stored in the **Aspect** value holder whenever the button is selected.
5. Apply the properties and install the canvas, and use **define** to add an instance variable and accessor method for the aspect.

6 Create an initialize method, in which you initialize the aspect variable to hold a value holder containing one of the valid **Select** symbols.

The default symbol determines which radio button is the default.

```
initialize
    super initialize.
    outputMode := #dialog asValue.
    showMinutes := true asValue.
    showHours := true asValue.
    showSeconds := true asValue.
```

Radio Button Events

A Radio Button triggers events when its label is changing, when it gains and looses focus, and when the selection is turned on or off.

Radio Buttons send the #labelChanging, #labelChanged, #turnedOn and #turnedOff events only after the Radio Button has been created and displayed on the canvas. Thus, a Radio Button that has its value initially set to be turned on or off, will not trigger that event, since that value is assigned before the Radio Button is initially displayed on the canvas.

#labelChanging
When a radio button's label is about to change, the radio button triggers the #labelChanging event.

#labelChanged
After a radio button's label has changed, the radio button triggers the #labelChanged event.

#clicked
When a radio button is clicked on by the mouse, without regard to the on or off state of the radio button, the radio button triggers the #clicked event.

#gettingFocus
When a radio button receives focus, either by being tabbed to or clicking on, the radio button triggers the #gettingFocus event.

#losingFocus
When a radio button loses focus, either by being tabbed away from or by another widget on the canvas gaining focus, the radio button triggers the #losingFocus event.

#tabbed
When a radio button has focus, and the Tab key is pressed causing the focus to move to the next widget in the tab order, the radio button triggers the #tabbed event.

#backTabbed

When a radio button has focus and the Back-Tab (Shift-Tab) key is pressed causing focus move to the previous widget in the tab order, the radio button triggers the #backTabbed event.

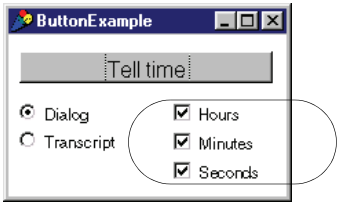
#turnedOn

After a radio button has its state changed from being not selected to being selected, the radio button triggers the #turnedOn event.

#turnedOff

After a radio button has its state changed from being selected to not being selected, the radio button triggers the #turnedOff event.

Check Boxes



check boxes

A check box is like a toggle button that allows the user to turn on or turn off an attribute. Check boxes are often used in a group to represent a set of related attributes.

Selecting one check box has no effect on others in the set, so users can select as many as they want. When you want only one attribute to be selected at a time, use radio buttons instead.

To add an check box:

1. Add a check box to the canvas for each selectable option.
2. In its **Label** property, enter a description for the check box.
3. Enter an **Aspect** property for the check box.

   The value holder for the check box will contain true when the check box is selected and false when it is not selected.
4. Apply the properties and install the canvas, and use **define** to create an instance variable and aspect accessor method.
5. Create or edit the initialize method to initialize the variable to a value holder containing true if you want the check box to be selected by default and false otherwise.

```
initialize
    super initialize.
    outputMode := #dialog asValue.
    showHours := true asValue.
    showMinutes := true asValue.
    showSeconds := true asValue.
```

Checkbox Events

A Check Box triggers events when its label is changing, when it gains and looses focus and when the selection is checked or unchecked.

Check Boxes send the #labelChanging, #labelChanged, #checked and #unchecked events only after the Check Box has been created and displayed on the canvas. Thus, a Check Box that has its value initially set to be checked or unchecked, will not trigger that event, since that value is assigned before the Check Box is initially displayed on the canvas.

#labelChanging
When a check box's label is about to change, the check box triggers the #labelChanging event.

#labelChanged
After a check box's label has changed, the check box triggers the #labelChanged event.

#clicked
When a check box is clicked on by the mouse, without regard to the checked or unchecked the check box, the check box triggers the #clicked event.

#gettingFocus
When a check box receives focus, either by being tabbed to or by clicking on by the mouse, the check box triggers the #gettingFocus event.

#losingFocus
When a check box loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the check box triggers the #losingFocus event.

#tabbed
When a check box has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the check box triggers the #tabbed event.

#backTabbed
When a check box has focus, and the Back-Tab (Shift-Tab) key is pressed in request to have the focus move to the previous widget in the tab order, the check box triggers the #backTabbed event.

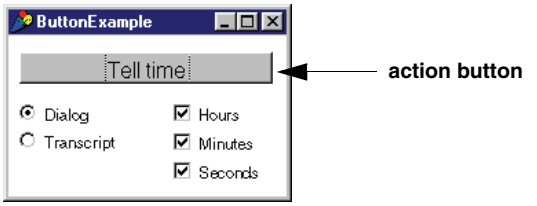
#checked

After a check box has had its state changed from being unchecked to being checked, the check box triggers the #checked event.

#unchecked

After a check box has its state changed from being checked to being unchecked, the check box triggers the #unchecked event.

Action Buttons



An action button triggers an action, such as opening a dialog window. If you want to save space in an interface, consider using a menu instead of multiple buttons.

To add an action button:

1. Add an action button to the canvas. In the button's **Label** property, enter a descriptive label.
2. In the button's **Action** property, enter the message selector that performs the action.
3. Apply the properties and install the canvas.
4. Create the method for the button in the actions protocol.

```
tellTime
        | t tString |
        t := Time now.
        tString := String new.

        "Assemble the time string based on the check boxes."
        self showHours value
            ifTrue: [tString := tString, t hours printString].
        self showMinutes value
            ifTrue: [tString := tString, ':', t minutes printString, ':']
            ifFalse: [tString := tString, '::'].
        self showSeconds value
            ifTrue: [tString := tString, t seconds printString].

        "Send the time string to the output channel set by the radio
         buttons."
        self outputMode value == #transcript
            ifTrue: [Transcript show: tString; cr]
            ifFalse: [DialogView warn: tString]
```

Default Button

An Action Button can be set as the default action when **Enter** is pressed, by checking the **Default** check box on the **Basics** page.

If the user presses **Enter** when either no widget has focus or the widget that has focus does not process **Enter**, then the default button is activated. If more than one button is set as default (which should be avoided in a GUI design), the first button in the tab order is the actual default. In any case, if a button has focus and processes **Enter**, it is the defacto “default” of the moment.

Note that setting a button as the default does not ensure that it will have focus when the UI opens.

In a subcanvas, if the subcanvas has a default button and has focus, its default button is the default. Otherwise, the application’s default button, if any, is the default.

Action Button Events

An Action Button triggers events when its label is changing, when it is pressed, and when it gains and looses focus.

Action Buttons send the #labelChanging and #labelChanged events only after the Action Button has been created and displayed on the canvas. Thus, an Action Button does not trigger these events when it gets its initial label, since that value is assigned before the Action Button is initially displayed on the canvas.

#labelChanging

When a button's label is about to change, the button triggers the #labelChanging event.

#labelChanged

After a button's label has changed, the button triggers the #labelChanged event.

#clicked

When a button is clicked on by the mouse, the button triggers the #clicked event.

#pressed

When a button is pressed either by keyboard interaction or by being clicked on by the mouse, the button triggers the #pressed event.

#gettingFocus

When a button receives focus, either by being tabbed to or by clicking on by the mouse, the button trigger the #gettingFocus event.

#losingFocus

When a button loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the button triggers the #losingFocus event.

#tabbed

When a button has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the button triggers the #tabbed event.

#backTabbed

When a button has focus, and the Back-Tab (Shift-Tab) key is pressed in request to have the focus move to the previous widget in the tab order, the button triggers the #backTabbed event.

Button Tab Notebook

The Button Tab Notebook widget was “soft” deprecated in 5i.4, and is no longer on the Palette. The supporting classes are still present in the system, but will be removed in a subsequent release. Please consider using Tab Control in its place.

Charts

The Chart widget supports a large variety of graphical representations that are commonly used in business applications to represent numerical data. The style and appearance of the chart are set by widget properties.

The application model provides the chart data in the form of a collection of numerical data. A simple chart presents a single set or series of values. A more complex chart, comparing two or more sets of data, requires two or more collections of values, called a *data series*.

Loading the Chart Widget

By default, the chart widget is not loaded or installed in the Palette. To make chart widgets available to the UI Painter, load the BGOK parcel. The parcel also includes example classes referred to in this section.

Adding a Chart

To add a bar chart to an application:

1. Select the Chart widget in the Palette, and drop it on a canvas.
2. On the **Basics** property page, specify the chart's **Aspect**, which is the the name of the instance variable and accessor method that supplies the data to the chart.

   For a simple chart with a single set of data, the aspect returns a collection containing those values. For charts that compare multiple data series, and charts with labels as well as values, the aspect returns a collection or items with additional structure, as described in "Charting Multiple Data Series" below.
3. Select the chart **Type:**.

   Chart types are described under "Chart Types" below.
4. **Apply** and **Install** the canvas.
5. **Define** the chart widget's aspect and edit it to return a ValueHolder on the collection containing the chart data.

Charting Multiple Data Series

It is frequently necessary to chart multiple data series, to provide a graphical comparison of values, such as a comparison of performance of two or more entities over time.

The **Aspect** for your chart, declared on the **Basics** property page, is still the instance variable for your chart data, but it now holds a collection of items with some additional structure.

In the simplest case, the aspect is a collection of collections of numbers, such as the following array of arrays:

```
#(#(60 40) #(20 55) #(35 10) #(75 50))
```

This defines two data series, which can be identified by index within the subcollections For example, in BG_EzChart. the **Aspect** is specified as **#data2**, which specifies the collection in its accessor method:

```
data2
    ^#(#(100 20 'A')  #(50 30 'B') #(-30 35 'C') #(34 30 'D') #(50 70 'E'))
```

On the **Data Series** property page, add two new **Data Set:** items, giving each a name, for example **Data1** and **Data2**. By default, with the aspect field left empty, items are assigned to the data sets in order, so Data1 is assigned #(100 50 -30 34 50) and Data2 is assigned #(20 30 35 30 70). In this case the aspect can simply be a collection, rather than a value holder on the collection. The label is added by defining an additional data set, as described under "Adding Labels" below.

If the data in the subcollections are not in the correct order, you can change the order by specifying the index as the data set aspect. In this case, the aspect must be a ValueHolder on the collection.

More typically, the collection will consist of structured objects defined by a class other than the application model, such as the Employee object used in the BG_Company example. In this case the data series uses accessor methods defined in the relevant class to return the collections required for charting.

1. Add a Chart widget to a canvas.
2. On the **Basics** property page for the chart, specify the **Aspect** for the chart, which returns a value holder on a collection.
3. On the **Data Series** page, click **New** to add a new **Data Set**, and
   - In the **Name** field, enter a descriptive name for the first data set.
   - In the **Aspect** field, enter the accessor method selector for the items making up this data set, or the index of this item in the subcollection. The accessor method will be defined in class defining the collection's elements.
4. Repeat step 3 for each additional data set.
5. **Apply** your changes.

6 Using a browser, ensure that the Chart widget's aspect instance variable is initialized to a ValueHolder on a collection.

```
initializeData
    "Initalize the employees collection."

    |employee|
    employee := BG_Employee new.
    employee name: 'Fred'; age: 50; salary: 40.
    employees  add: employee.
    employee := BG_Employee new.
    employee name: 'Joe'; age: 25; salary: 32.
    employees add: employee.
    employee := BG_Employee new.
    employee name: 'Bob'; age: 36; salary: 33.
    employees  add: employee.
```

7 If necessary, define the accessor methods for each data set added in step 3 (as in BG_Employee).

Adding Labels

To add labels to a chart, include the label strings as another data set.

To add labels:

1 On the **Data Series** property page, add a new data set with an appropriate name. For the **Aspect**, enter the index for the labels in the collection, or the name of the accessor method that returns the label from the collection members.

2 **Apply** your changes.

3 Edit the application code as necessary to include the label in the collection stored in the chart widget's aspect instance variable.

Chart Properties

Basic

Aspect

The message selector that returns the collection containing the chart data and labels.

ID

The identifying name for the chart widget.

Type

The type of chart (see descriptions below under "Chart Types").

Orientation
Specifies whether the data scale is oriented vertically or horizontally.

Options

The properties on the Options page vary widely depending on the specific chart type, and so are covered for individually for each chart type in the sections below.

Data Series

Data Set
A drop-down list of the data sets defined for this chart. To add a new data set, click **New** and provide a name and other parameters. To remove a data set, select the data set then click **Delete**.

Name
The identifying string for the data set.

Aspect
Either:

- Blank, in which case data points are taken from member collections in the order the data sets are listed.
- An integer, in which case data points are taken from member collections as specified by the index.
- A method selector, in which data points are retrieved by sending the corresponding access message.

Label
If checked, the data set is interpreted as providing a label for the data sets.

Computed
If checked, the model is expected to be computed within the application model, typically using a PluggableAdaptor, rather than represented as an AspectAdaptor on the collection element (see BG_StockTool for an example).

Color…
Set foreground and background colors.

Lines…
Set line style and weight.

Pattern…
Set fill pattern.

Legend

This properties page allows you to select the location of the chart legend, relative to the chart itself, or to display no legend (**hidden**).

Item - Axis

For charts with a vertical orientation (**Basics** page), the item axis runs horizontally.

Title
: A general label for the data items, displayed below the data labels, if any.

Units
: A label to describe the units for the data items.

Section
: Select placement of lines to divide display into sections. Select **None** to hide lines, **Major** to place lines at data points, **Division** to place lines between data points, **Edge** to place a line at the end of the chart, **Zero** to place a line at the zero end of the chart.

Ticks
: Placement of tick marks. Select **Hide** to hide tick marks, **Cross** to cross the boundary, **Inside** to place ticks inside the chart boundary, **Outside** to place ticks outside the chart boundary.

Border
: If checked, includes a line at the data border.

Axis
: If checked, includes a line along the axis line.

Item - Scale

Auto Scale
: Set and disabled for all charts except XY charts. When unchecked (XY charts only), allows you to set the **Zero**, **Div**, **Min**, **Max**, **Step**, and **Scale Type** parameters.

Invert Scale
: Inverts the chart: horizontally, if oriented vertically on the **Basics** page.

Zero
: The value at which **Step** increments are started, and the low end boundary line displayed. For most purposes, this value should be less than the **Min** value.

Div
: The number minor divisions marked by tick marks between major division lines.

Min
The value at the low end of the chart scale.

Max
The value at the high end of the chart scale.

Step
Increment between major division lines

Scale Type
Either **Normal** or **Log** (logarithmic). For a logarithmic scale, specify the **Base**.

Data - Axis

For charts with a vertical orientation (**Basics** page), the data axis runs vertically.

Title
A general label for the data scale, displayed left of the scale labels.

Units
A label to describe the units for the data scale.

Section
Select placement of lines to divide display into sections. Select **None** to hide lines, **Major** to place lines at data points, **Division** to place lines between data points, **Edge** to place a line at the end of the chart, **Zero** to place a line at the zero end of the chart.

Ticks
Placement of tick marks. Select **Hide** to hide tick marks, **Cross** to cross the boundary, **Inside** to place ticks inside the chart boundary, **Outside** to place ticks outside the chart boundary.

Border
If checked, includes a line at the data border.

Axis
If checked, includes a line along the axis line.

Format
Numeric format for the displayed labels:

Position
Location of numeric labels.

Data - Scale

Auto Scale
Set and disabled for all charts except XY charts. When unchecked (XY charts only), allows you to set the **Zero**, **Div**, **Min**, **Max**, **Step**, and **Scale Type** parameters.

Invert Scale
Inverts the chart: horizontally, if oriented vertically on the **Basics** page.

Zero
The value at which **Step** increments are started, and the low end boundary line displayed. For most purposes, this value should be less than the **Min** value.

Div
The number minor divisions marked by tick marks between major division lines.

Min
The value at the low end of the item scale.

Max
The value at the high end of the item scale.

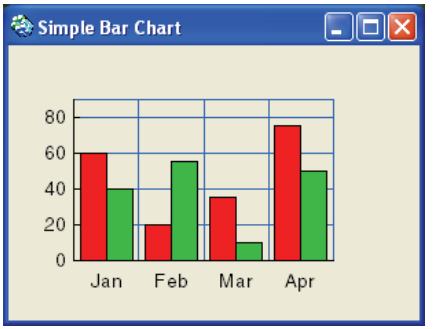
Step
Increment between major division lines

Scale Type
Either **Normal** or **Log** (logarithmic). For a logarithmic scale, specify the **Base**.

Chart Types

This section briefly describes the charts support by the Chart widget, and any special requirements for those charts.

Bar Chart



A standard bar chart uses the length of its bars to represent the magnitude of its data. Bars for data points for multiple data series are displayed side-by-side. The data can include labels.

The **Options** page properties are:

Bar Width
A scale to set the bar width from thin to wide enough to fill the chart.

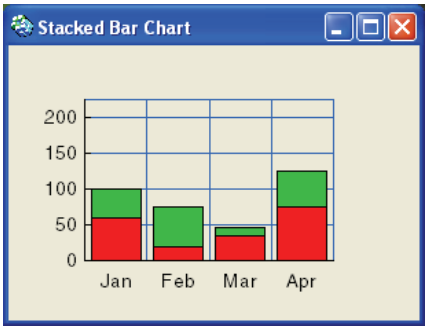
Bar Overlap
A scale to set the amount of overlap or gap. Centered, the bars touch. To the right, earlier data set bars overlap later bars. To the left, a gap is left between bars.

Data Location
Where bar data is displayed. **None**, no data value is shown; **Inside**, data value is in the bar; **Outside**, data value is above the bar.

Link Line
Not used for Bar Charts.

Picture
Not used for Bar Charts.

Stacked Bar



A stacked bar chart compares both the total amount and magnitude of each item by stacking the data series bars on top of another.

```
#(#(60 40 'Jan') #(20 55 'Feb') #(35 10 'Mar') #(75 50 'Apr'))
```

The **Options** page properties are:

Bar Width
A scale to set the bar width from thin to wide enough to fill the chart.

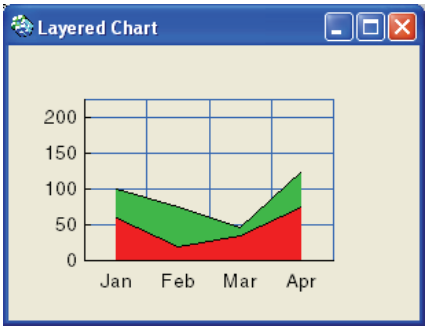
Bar Overlap
Not used for Stacked Bar charts.

Data Location
Where bar data is displayed. **None**, no data value is shown; **Inside**, data value is in the bar; **Outside**, data total is above the bar.

Link Line
Connects data set levels with connecting lines.

Picture
Not used for Stacked Bar charts.

Layer



A layer chart indicates the change in the total amount and the individual amounts of the items within a specified time period. Like a stacked bar chart, each data series is placed on top of another. A layer chart, however, connects each item together, denoting the change from item to item.

The **Options** page properties are:

Bar Width
Not used for Layer Charts.

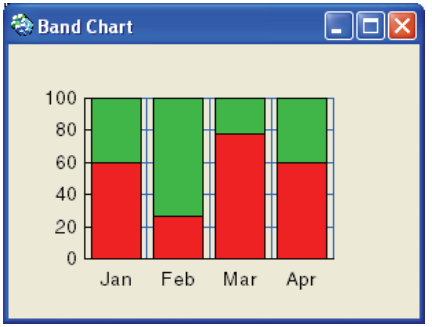
Bar Overlap
Not used for Layer Charts.

Data Location
Where data is displayed. **None**, no data value is shown; **Inside**, data value is in the area; **Outside**, shows data totals above the area.

Link Line
Not used for Layer Charts.

Picture
Not used for Layer Charts.

Band



A band chart indicates the relative amounts of the components of an item.

The **Options** page properties are:

Bar Width
A scale to set the bar width from thin to wide enough to fill the chart.

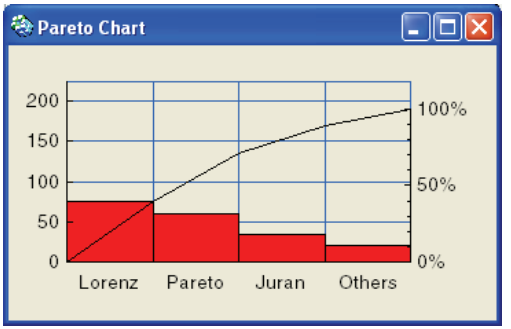
Bar Overlap
Not used for Band charts.

Data Location
Where bar data is displayed. **None**, no data value is shown; **Inside**, data value is in the bar; **Outside**, no data value is shown.

Link Line
Connects data set levels with connecting lines.

Picture
Not used for Band charts.

Pareto



A Pareto chart consists of a bar chart, with the bars arranged in descending order of item magnitude, with an overlay graph of cumulative percentages. It is useful for identifying the relatively few contributors that account for the bulk of an effect (the so-called “80-20 rule”).

The Pareto chart is typically used for a single data series, but can take multiple data series.

The figure above uses the following as its data:

```
#(#(60 'Pareto') #(20 'Others') #(35 'Juran') #(75 'Lorenz'))
```

Notice that the order of the data is not maintained in the chart, which shows the data points in decreasing order of magnitude.

The **Options** page properties are:

Bar Width
A scale to set the bar width from thin to wide enough to fill the chart.

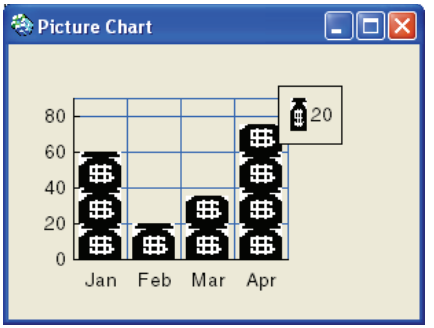
Bar Overlap
Not used for Pareto charts.

Data Location
Where bar data is displayed. **None**, no data value is shown; **Inside**, data value is in the bar; **Outside**, data total is above the bar.

Link Line
Not used for Pareto charts.

Picture
Not used for Pareto charts.

Picture



A picture chart represents the unit of a numeric item with a meaningful picture or symbol. It indicates the magnitude of an item with the number of pictures. A fraction of the picture is used when the amount does not equal an even multiple of pictures. As shown in the chart legend above, each picture represents twenty dollars. In March, the amount is thirty-five dollars, so a portion of the picture is show to indicate this amount.

A picture chart can only represent one data series, plus a set for labels. The above figure uses the following as its data:

```
#(#(60 'Jan') #(20 'Feb') #(35 'Mar') #(75 'Apr'))
```

By default, a Picture Chart displays a legend, which can be moved or hidden on the **Legend** property page.

The **Options** page properties are:

Bar Width
A scale to set the bar width from thin to wide enough to fill the chart.

Bar Overlap
Not relevant for a single data series.

Data Location
Where bar data is displayed. **None**, no data value is shown; **Inside**, data value is in the bar; **Outside**, data value is above the bar.

Link Line
Not used for Picture Charts.

Picture/Aspect
The method selector that returns the graphic to use as the chart picture. The graphic can be an Image or subclass, or an OpaqueImage. For example, with two class resource methods created using the Image Editor, the accessor method could be:

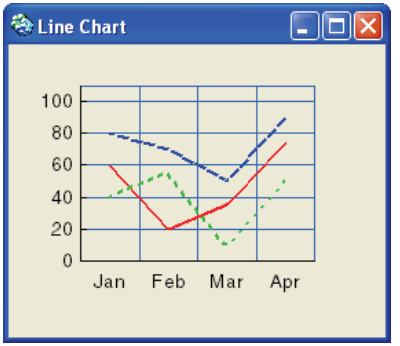
```
coins
    ^(OpaqueImage figure:
        self class coins
        shape: self class coinsMask)
```

Note that the chart implementation does not use the mask, so background is not transparent.

Picture/Unit

Set the number of units represented by a complete picture. By default, the picture represents a unit of 20.

Line



A line chart indicates the change in the magnitude of data over a period of time.

The figure above uses the following as its data:

```
#(#(60 40 80 'Jan') #(20 55 70 'Feb') #(35 10 50 'Mar') #(75 50 90 'Apr'))
```

The **Options** page properties are:

Line

Either **Show** or **Hide** the line between data points. The **Spline** check box makes the line a spline (curved).

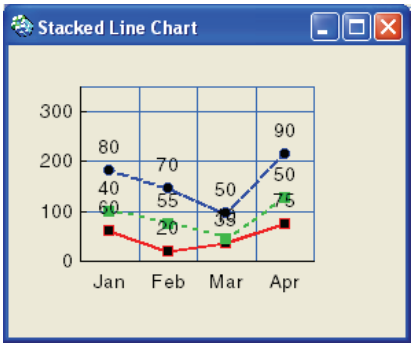
Marker

Either **Show** or **Hide** a marker for the data points.

Data

Either **Show** or **Hide** data values above the data points.

Stacked Line



A stacked line chart compares the total amount of items with the magnitude of the components of the items.

The stacked line chart above uses the same data as the line chart.

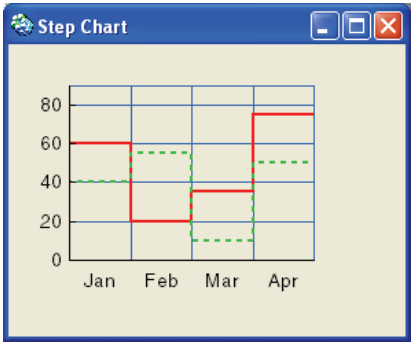
The **Options** page properties are:

Line
Either **Show** or **Hide** the line between data points. The **Spline** check box makes the line a spline (curved).

Marker
Either **Show** or **Hide** a marker for the data points.

Data
Either **Show** or **Hide** data values above the data points.

Step



Like a regular line chart, a step chart indicates the change in the magnitude of data over time. It is useful when the changes of magnitude between intervals are of interest.

The step chart above uses the data:

```
#(#(60 40 'Jan') #(20 55 'Feb') #(35 10 'Mar') #(75 50 'Apr')
```

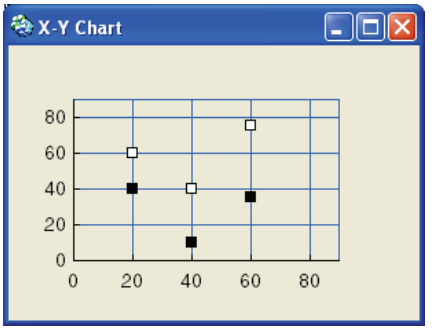
The **Options** page properties are:

Line
Either **Show** or **Hide** the line between data points. The **Spline** check box makes the line a spline (curved).

Marker
Either **Show** or **Hide** a marker for the data points.

Data
Either **Show** or **Hide** data values above the data points.

X-Y Chart



An XY chart indicates any correlations between two characteristics derived from the scattering status of points plotted in the chart.

The XY chart expects points as its data. For example, an XY chart with two data series could be set up as follows:

```
dataPointXY

    | array1 array2 array3|

    array1 := Array with: 20@60 with: 20@40.
    array2 := Array with: 40@40 with: 40@10.
    array3 := Array with: 60@75 with: 60@35.

    ^(Array with: array1 with: array2 with: array3) asValue
```

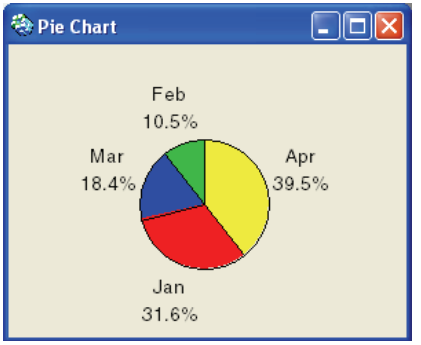
The **Options** page properties are:

Line
Either **Show** or **Hide** the line between data points. The **Spline** check box makes the line a spline (curved).

Marker
Either **Show** or **Hide** a marker for the data points.

Data
Either **Show** or **Hide** data values above the data points.

Pie Chart



A pie chart can only represent one data series, but may include labels.

The chart above uses the following data:

```
#(#(60 'Jan') #(20 'Feb') #(35 'Mar') #(75 'Apr'))
```

The **Options** page properties are:

Exploding Labels
In a pie chart, a slice can be designated to be exploded. For example, enter **Jan** in the **Labels to Explode** field. Only one label can be exploded at a time.

Doughnut Labels
A doughnut label appears inside a circle in the center of a pie chart. To add a doughnut label, enter the label name in the **Doughnut Label** field. This label is separate from the labels included as a data set.

Data
Location of the data value relative to the label. **Hide**, no data shown; **Below**, below the label; **Adjacent**, same line following the label.

Shape
Chart shape. Either **Circle** or **Ellipse**.

Lines
Lines connecting lable/data with a slice of the pie. Either **Show** or **Hide**.

Click Map

The Click Map widget allows you to define areas of a graphic image and specify the action to invoke if the user clicks on each area.

Using a Click Maps for a button bar makes it possible to use any graphic image to represent the row of buttons. The image may include text, graphics, and a variety of colors. All of the buttons may be part of a single image, guaranteeing their relative layout.

Adding a Click Map

To add a click map widget to a canvas:

1. Choose the click map widget  in the Palette and place it on a canvas.
2. Set the following **Basic** properties and **Apply** them:

   ID: A unique identifier for the widget, which will be used to access the widget.

   Visual Message: The message selector that returns the image, which is typically an image resource method.

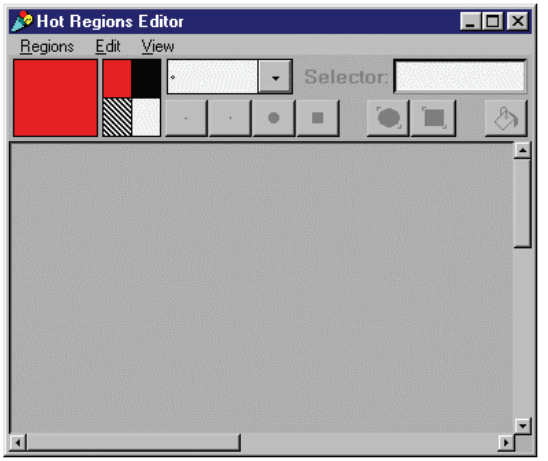
   Mappings Selector: The message that returns the mapping between an area and the action for that location. You define mapping using the Hot Regions Editor.

   Default Click Message: (optional) Specifies the action to be invoked when the user clicks in an area for which no other action is specified. If left empty, no action is taken.
3. In the canvas, resize the click map widget so that it closely outlines the graphic image.
4. **Install** the canvas, and provide a class name and method name in which to install the canvas when prompted.

Defining the Hot Region Mappings

1. Select the click map widget in the canvas.

2 In the GUI Painter Tool, choose **Tools → Hot Regions Editor**.



3 Choose **Regions → Read** to display the selected click widget's image as the background in the Hot Regions Editor. Resize the window to display the entire image.

4 Choose **Edit → New Slice** to begin defining a region.

A slice is comprised of one or more areas that invoke the same action when clicked. The areas in a slice do not need to be contiguous. Identify areas in the Hot Regions Editor using tools similar to other bitmap editors:

- The four colors or patterns of ink allow you to select a color that shows well when selecting a region. The color does not show in the final click map.
- Six brush sizes and shapes. The first four draw lines, the next draws an ellipse, and the sixth draws a rectangle.
- A fill-mode bucket that fills the entire image.

5 In the **Selector** input field, enter a method selector for the action to be invoked when this slice is clicked and press <Return>.

The selector is added to the menu button in the middle of the Hot Regions Editor. You can use the menu to switch between different slices of the same hot region resource.

6 Repeat steps 4 and 5 to define all the slices for this hot region.

7 Choose **Regions → Apply** to:

- Install the hot region resource. You are prompted for the class name and selector for the resource method that stores the hot region mappings.
- Insert the resource's selector into the **Mapping Selector** property of the click map widget that is selected in the canvas.

8 Close the Hot Regions Editor.

9 Install the canvas.

Using Custom Views and Controllers

If you have an application that uses custom views and controllers, you can make it work as a VisualWorks application:

1 Make the custom view a subclass of ClickWidget.

2 In the custom view, implement a mouseReleaseAt: method that specifies what to do when the user clicks in the custom view.

3 Disconnect the custom controller from your application. Move any custom behavior to the custom view's mouseReleaseAt: method.

Note that interaction with the user is more limited in web applications than in other applications. In particular, the only mouse events transmitted to the application from the web browser are mouse clicks. Events based on entry, mouse movement, and exit are ignored.

Click Widget Events

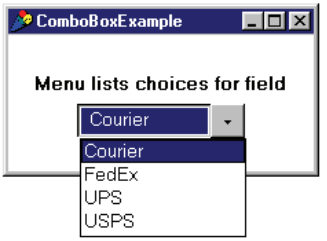
A Click Widget triggers events when its view is clicked on.

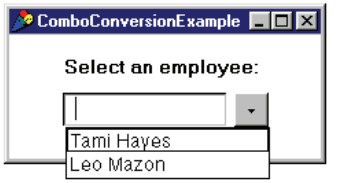
#clicked
When a click widget is clicked on by the mouse, without regard to if a mapped region is hit or not, the click widget triggers the #clicked event.

#hitMappedRegion
When a click widget is clicked on by the mouse, and at least one of the mapped regions of the click widget has been hit, the click widget triggers the #hitMappedRegion event.

#missedMappedRegion
When a click widget is clicked on by the mouse, and none of the mapped regions of the click widget has been hit, the click widget triggers the #missedMappedRegion event. If a click widget has a **Default Click** selector defined, then the click widget will never trigger the #missedMappedRegion event.

Combo Box





A ComboBox presents a list of options and allows the user to select one. This is often a more useful interface when a limited number of options are acceptable than is an input field with validation on entries.

The ComboBox has both a collection of options and an item selected in that collection.

Adding a ComboBox to a Canvas

Online example: ComboBoxExample

1. Add a combo-box widget to the canvas.
2. On the Basics property page, in the **Aspect** field, enter a name for the aspect accessor method (in the example, **shipper**). This method returns a value model on the selected item for display.
3. In the **Choices** property, enter the name of the method that returns value model on a collection of entry choices (**shipperChoices**).
4. Apply the properties and install the canvas, and use **Define** to create the aspect instance variable and accessing method.
5. Create the ComboBox choices method in the application model.

   The method returns a value holder containing the list of valid entries. The value holder can be held in an instance variable (as in the example).

   ```
   shipperChoices
       ^shipperChoices
   ```

6. Initialize the field's aspect variable with a value model containing data of the type specified in the **Type** property for the widget.
7. Initialize the choices variable with a value holder containing the list of valid entries.

```
initialize

    | list |
    shipper := 'Courier' asValue.

    list := List new.
    list add: 'Courier';
    add: 'FedEx';
    add: 'UPS';
    add: 'USPS'.
    shipperChoices := list asValue.
```

Listing Arbitrary Objects

You can arrange for a combo box to display a list of choices that are arbitrary objects (for example, a list of Employee objects). You do this by supplying a *print method* and a *read method* that translate the relevant objects into displayable elements (for example, Strings or graphical images) and back. For example, in ComboCoversionExample, the print method enables the combo box to display Employee names in the pull-down list and, when an Employee is selected, to display that employee's name in the field. The read method enables the combo box to interpret the user's input as an Employee name, which can be matched with an existing Employee, or used to create a new one.

Online example: ComboConversionExample

1. On the Basics property page, set the **Type** property of the combo box to **Object**.
2. Fill in the **Print** property with the name of a method for converting the relevant objects to strings (in this example, employeeToString:). The name must end with a colon.
3. Fill in the **Read** property with the name of a method for converting strings to objects of the desired type (in this example, stringToEmployee:). The name must end with a colon.
4. In the application model, create a print method with the name you specified in step 2 (employeeToString:). This method accepts an object from the choices list as an argument (in this case, an instance of Employee).
5. In the print method, return a String that represents the object from the choices list. In this example, display the name of the Employee. The string is displayed in the combo box's pull-down list and also in the combo box's field when the choice is selected.

```
employeeToString: anEmployee

"Return a String for representing the Employee in the combo box's list
and field."

    ^anEmployee name.
```

6 Create a read method with the name you specified in step 3 (stringToEmployee:). This method accepts a String argument.

7 In the read method, return an object for the given String. In this example, determine whether the String is the name of an Employee in the choices list; if so return that Employee. Otherwise, create a new Employee and add it to the choices list.

```
stringToEmployee: aString
"Return an Employee corresponding to the given String. If the String
corresponds to the name of an Employee on the choices list, return that
Employee. Otherwise, create a new Employee and add it to the list."

    | theEmp |
    theEmp := self employeeChoices value
                detect: [:each | each name = aString]
                ifNone: [nil].

    theEmp isNil
        ifTrue:
            [theEmp := Employee new name: aString.
            self employeeChoices value addLast: theEmp].

    ^theEmp
```

Combo Box Events

A Combo Box triggers events when a selection is changing, when it gains and looses focus, when it gets clicks from the mouse, and when its list is exposed and closed. The biggest difference between a Menu Button and a ComboBox in terms of events triggered, is that a Menu Button does not trigger the #rightClicked or #doubleClicked events.

#changing

When a selection in the list of a combo box is made, or in the case of an editable combo box, the value has been edited, but before the value is accepted, the combo box triggers the #changing event.

#changed

After a selection in the list of a combo box is made, or in the case of an editable combo box, the value has been edited, when the value is accepted, the combo box triggers the #changed event. To trigger #changed at each keystroke, send continuousAccept: to the widget's controller with the value true.

#clicked

When a combo box is clicked on with the <Select> mouse button, the combo box triggers the #clicked event.

#rightClicked

When a combo box is right clicked on with the <Operate> mouse button, the combo box triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the combo box.

#doubleClicked

When a combo box is double clicked on with the <Select> mouse button, the combo box triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event. Read-only combo boxes do not respond to the double clicked event, but rather consider the double click to be two separate clicks that open and immediately close the combo box, and thus do not trigger the #doubleClicked event.

#gettingFocus

When a list box receives focus, either by being tabbed to or by clicking on by the mouse, the list box triggers the #gettingFocus event.

#losingFocus

When a combo box loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the combo box triggers the #losingFocus event.

#tabbed

When a combo box has focus, and the Tab key is pressed moving the focus to the next widget in the tab order, the combo box triggers the #tabbed event.

#backTabbed

When a combo box has focus, and the Back-Tab (Shift-Tab) key is pressed moving the focus to the previous widget in the tab order, the combo box triggers the #backTabbed event.

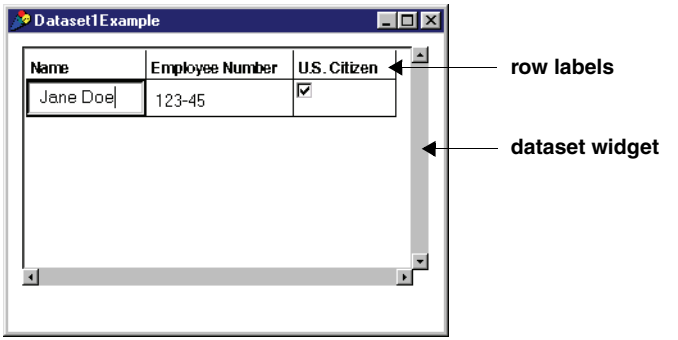
#listExposed

When a combo box has its selection list exposed, either by clicking on the combo box or using the down arrow key to open the list, the combo box triggers the #listExposed event.

#listClosed

When a combo box has its selection list closed, either by clicking on another widget on the canvas, by clicking on the combo box when the list is already exposed, or by selecting a new value in an exposed list, the combo box triggers the #listClosed event.

Datasets



A dataset presents a list of similar objects for a user to edit. Datasets are similar to tables, but cells in a dataset can be edited directly. Datasets are best suited for presenting similar kinds of data.

A dataset uses a SelectionInList to hold the list of objects to be displayed, along with information about the current selection. Each object in the list is displayed in its own row, with individual aspects of the object displayed in their own columns. The property pages specify the means by which each column presents its data, whether using cells that contain read-only fields, editable fields, combo boxes, or checkboxes.

Setting up a Dataset

Online example: Dataset1Example

The example displays instances of Employee, which consist of three objects (name, empNo, and citizen), in three dataset columns.

1. Add the dataset widget to the canvas.
2. On the **Basics** property page, in the dataset's **Aspect** property, enter the aspect name (dsvList). **Apply** the property and install the canvas.
3. Use **define** to add the dsvList instance variable and accessor method to the application model.

   The dsvList method returns a SelectionInList object that will hold the list to be displayed. This method also sets up the SelectionInList so it will cause a user's selection to be put in a separate value holder (selectedRow).
4. In the dataset's properties, click the **New Column** button for each column you want in the dataset.

In the example, three columns are added to the canvas.

5 In the canvas, <Alt>-click in the leftmost column to select it.

6 Display the dataset's **Column** property page. The properties you set on this page will apply to the currently selected column.

7 On the **Column** page, enter **Name** as the **Label** property. This creates a visual label above the selected column, which is to display employee names.

8 On the **Column** page, enter selectedRow name in the **Aspect** field. selectedRow refers to the value holder that will hold the object (the Employee) selected by the user. name refers to the aspect of Employee that is displayed in this column.

9 On the **Column** page, select **Input Field** as the **Type**. This causes each cell in the selected column to display its data in an editable input field. Note that you can optionally specify nondefault characteristics for these input fields by filling in properties on the **Column Type** page.

10 <Alt>-click on the middle column to select it.

11 On the **Column** page, enter **Employee Number** as the **Label** and selectedRow empNo as the **Aspect**. Select **Input Field** as the **Type**.

12 <Alt>-click on the rightmost column to select it.

13 On the **Column** page, enter **U.S. Citizen** as the **Label** and selectedRow citizen as the **Aspect**. Select **Check Box** as the **Type**.

14 When the all properties have been applied, install the canvas.

15 Use the **define** command to add the selectedRow instance variable to the application model and to create the selectedRow method in the aspects protocol.

The selectedRow method returns a value holder for holding the user-selected Employee object from the SelectionInList.

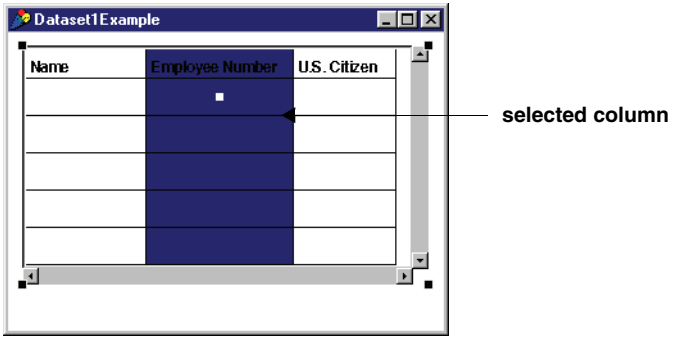
16 Use a browser to initialize the dataset (in an initialize method in an initialize-release protocol).

```
initialize
    | aList |
    aList := List new.
    aList add: Employee new initialize.
    self dsvList list: aList.
```

When you open the application, the dataset contains one empty row. You can type a name and number in the **Name** and **Employee Number** columns, and select the **U.S. Citizen** check box.

Note that the first part of the **Aspect** setting for each column must be the same as the message sent by the SelectionInList to obtain a value holder for storing the selected object. You used selectedRow in steps 8, 11, and 13, because that name is used in the generated dsvList method that sets up the SelectionInList (step 3). To use a name other than selectedRow, you must replace selectedRow with the desired name in each **Aspect** field *and* in the code generated for dsvList in step 3. Use the **define** command (as in step 15) to generate an instance variable and method with the new name.

Editing Column Properties



You must select a column before you can set its properties. To select a column:

1. Select the dataset on the canvas.
2. Place the cursor inside one of the columns of the dataset.
3. Hold down the <Control> or <Alt> key while clicking the <Select> mouse button.

Changing Column Widths

By default, all columns have a width of 80 pixels. You can set specific widths in the dataset's **Column** properties. You can also change the column widths by editing the dataset in the canvas. For example, to resize the **Employee Number** column:

1. In the canvas, <Alt>-click in the **Employee Number** column to select it.
2. Place the cursor near the top right or top left margin of the column.

3 Click and hold the mouse button. The cursor changes to indicate a column width change operation when properly selected. If necessary, move the pointer nearer the corner of the selected column until the cursor changes appearance.

4 Drag the cursor to widen or narrow the column.

5 Install the canvas.

Changing the Column Order

You can switch the order of a dataset's columns by editing it in the canvas. For example, to switch the order of the **Employee Number** and **U.S. Citizen** columns:

1 In the canvas, <Alt>-click in the **Employee Number** column to select it.

2 Place the cursor on the drag handle within the selected column.

3 Click and hold on the handle, and drag it toward the **U.S. Citizen** column. The cursor changes to indicate the move operation when properly selected.

4 Install the canvas.

Disabling Column Scrolling

You can set a dataset's columns so that they cannot be scrolled horizontally. This is useful if you want to keep one or more columns displayed on the dataset at all times, while the others continue to scroll.

1 To disable scrolling for a column (and all columns to the left of it), select that column and click the **Fixed** check box in the **Column** properties.

2 Apply the property and install the canvas.

Moving the Selection to Another Column

1 Select a column in the dataset using the basic steps.

2 Click the <Select> mouse button for subsequent column selections.

If you then select another widget on the canvas, you must repeat the basic steps to reselect a dataset column.

Scrolling Dataset Columns

You can scroll the columns in the dataset you are painting:

1 Select a column in the dataset.

2 Press <Control> while using the mouse to move the scroll bars on the dataset.

Formatting Column Labels

When you specify a column label by entering a string in the **Label** property, you have no formatting control. You can add formatting to a label by using a ComposedText, which is a graphic, to set the label. (Refer to “Working with Text,” Chapter 15 in the *Application Developer’s Guide*.)

For example, to split a long label onto two lines, you provide a composed text that contains the appropriate carriage returns. (See Dataset4Example in the online examples.) To split the **Employee Number** column label:

1. Create a class method (number) in a resources protocol of the application model. This method returns a composed text that is to appear as the label.

   ```
   number
       ^('Employee Number' asText allBold) asComposedText
   ```

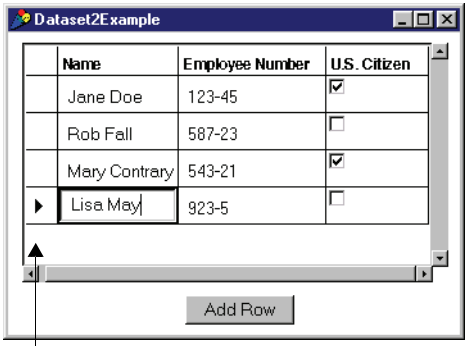
2. In the canvas, select the **Employee Number** column of the dataset.
3. On the **Basics** property page, enter number as the **Label** in the **Column** properties.
4. Select the **Image** check box next to **Label**. This specifies that the column label will come from the resource method named in the **Label** property.
5. Apply the properties and install the canvas.

Similarly, to change the color of the column label, set the text color in the number method:

```
number
    ^('Employee Number' asText emphasizeAllWith:
        (Array with: #bold with: #color->ColorValue red))
        asComposedText
```

Adding a Row

Online example: Dataset2Example



row marker

When the number of rows needed for a dataset is not predetermined, you can program your application to add rows while it is running.

1 Use the Palette to add an Action Button to a canvas containing a dataset. Leave the button selected.

2 On the **Basics** property page, enter **Add Row** as the button's **Label** property and addRow as the button's **Action** property. Apply the properties and install the canvas.

3 Using the **define** command or a System Browser, add the instance method addRow in the actions protocol. This method adds a new object to the list displayed by the dataset. This, in turn, adds a new row to the dataset.

```
addRow
    (dsvList list) add: Employee new
```

Adding a Row Marker

A row marker indicates which row is selected within a dataset. It is used in place of row highlighting. To add a row marker, check **Row Selector** on the dataset's **Details** properties. The marker appears as the first column within the dataset.

To make the row selector appear as a button, which may be useful to make it more obviously a clickable option, check **Row Selector As Buttons**, also. Both check boxes must be checked for this option to be active.

Adding Row Numbering

Rows can be automatically numbered by checking the **Show Line Numbers** property on the Details page. Numbers show in the same column with the row selection indicator.

Providing Initial Data

Online example: Dataset3Example

An initially empty dataset is sufficient if you want users to input the data after the application is open. However, some applications require their datasets to display data initially.

In the application model, create an initialize method that provides the data for your dataset.

```
initialize
    | aList anEmp |
    aList := List new.

    "The aspect for the dataset should be a list of Employee
    instances. Create an employee to put in the list."
    anEmp := Employee new initialize.
    anEmp name: 'Tami Hayes'; empNo: '341-2'; citizen: true.
    aList add: anEmp.

    "Create an employee to put in the list."
    anEmp := Employee new initialize.
    anEmp name: 'Leo Mazon'; empNo: '786-9'; citizen: false.
    aList add: anEmp.

    "Set the list for the dataset aspect. This list appears when you
    start."
    self dsvList list: aList.
    super initialize.
```

Dataset Properties

Traversal Page

The Traversal properties only appear for the DataSet widget, and consist of two sets of radio buttons. Traversal properties define the wrap action to be taken in the dataset when navigation keys are used:

Up	<Shift> <Up arrow>
Down	<Shift> <Down arrow>
Right	<Tab>
Left	<Shift> <Tab>

Vertical Policy/No Wrapping
Up and **Down** movement is supported, but does not wrap when at the top or bottom cell.

Vertical Policy/Wrapping
Up and **Down** movements are allowed, and wraps when at the top or bottom cell. At the top cell, an Up movement wraps to the bottom cell; at the bottom cell, a Down movement wraps to the top cell.

Vertical Policy/None
Ignore the **Up** or **Down** movement keys.

Horizontal Policy/No Wrapping
Left and **Right** movement is supported, but does not wrap when at the leftmost or rightmost column.

Horizontal Policy/Wrap to next row
Left and **Right** movement is supported, and wraps when at the leftmost or rightmost column. When at the leftmost column, moving left wraps to the rightmost column of the next higher row. When at the rightmost column, moving right wraps to the leftmost column of the next lower row.

Horizontal Policy/Wrap on same row
Left and **Right** movement is supported, and wraps on the same row.

Horizontal Policy/None
Ignore the Right or Left movement keys.

Dataset Events

A Dataset triggers events when a selection is changing, when it gains and looses focus, when it gets clicks from the mouse, and when it's view scrolls.

#rightClicked

When a dataset is right clicked on with the <Operate> mouse button, the dataset triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the dataset.

#doubleClicked

When a dataset is double clicked on with the <Select> mouse button, the dataset triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus

When a dataset receives focus, either by being tabbed to or by clicking on by the mouse, the dataset triggers the #gettingFocus event.

#losingFocus

When a dataset loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the dataset triggers the #losingFocus event.

#tabbed

When a dataset has focus and the last cell in the dataset has focus, and the Tab key is pressed to the focus to the next widget in the tab order, and the dataset has its wrapping policy that allows it to tab out to another widget, the dataset triggers the #tabbed event.

#backTabbed

When a dataset has focus and the first cell in the dataset has focus, and the Back-Tab (Shift-Tab) key is pressed to movethe focus to the previous widget in the tab order, and the dataset has its wrapping policy that allows it to tab out to another widget, the dataset triggers the #backTabbed event.

#scrollLeft

If while navigating with the keyboard or mouse, or manipulating a horizontal scroll bar in a dataset, the dataset scrolls to the left, the dataset triggers the #scrollLeft event.

#scrollRight

If while navigating with the keyboard or mouse, or manipulating a horizontal scroll bar in a dataset, the dataset scrolls to the right, the dataset triggers the #scrollRight event.

#scrollUp

If while navigating with the keyboard or mouse, or manipulating a vertical scroll bar in a dataset, the dataset scrolls up, the dataset triggers the #scrollUp event.

#scrollDown

If while navigating with the keyboard or mouse, or manipulating a vertical scroll bar in a dataset, the dataset scrolls down, the dataset triggers the #scrollDown event.

#cellGettingFocus

When an individual cell in a dataset receives focus, either by being tabbed to or by clicking on by the mouse, the dataset triggers the #cellGettingFocus event.

#cellLosingFocus

When an individual cell in a dataset loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the dataset triggers the #cellLosingFocus event.

#cellTabbed

When an individual cell in a dataset has focus, and the and the Tab key is pressed to move the focus either the next cell in the dataset or to the next widget in the tab order, the dataset trigger the #cellTabbed event.

#cellBackTabbed

When an individual call in a dataset has focus, and the Back-Tab (Shift-Tab) key is pressed in request to move the focus to either the previous cell in the dataset or the previous widget in the tab order, the dataset triggers the #cellBackTabbed event.

#cellValueChanged

If a cell is editable, and its value is changed, once the new value has been accepted the dataset triggers the #cellValueChanged event. This occurs when the value of the underlying input field changes, the selection of the underlying combo box changes, or the check or unchecked value of the underlying check box changes.

#columnLabelClicked

If a dataset has column labels, and that column label is clicked on with the mouse, then the dataset triggers the #columnLabelClicked event.

#rowLabelClicked

If a dataset has row labels, and the row label is clicked on with the mouse, then the dataset triggers the #rowLabelClicked event.

#rowSelectionsChanging

If a dataset has row labels and a row label is selected or unselected, or in the case of multiple select datasets the selections change, or in the case of a dataset where there are no row labels and the focus changes from one row to another, then prior to the selection change taking effect, the dataset triggers the #rowSelectionsChanging event.

#rowSelectionsChanged

If a dataset has row labels and a row label is selected or unselected, or in the case of multiple select datasets the selections change, or in the case of a dataset where there are no row labels and the focus changes from one row to another, then after the to the selection change has occured, the dataset triggers the #rowSelectionsChanged event.

#selectionListChanged

Whenever the list underlying a dataset is manipulated, either by changing a value, reordering, adding or removing items, or changing the list as a whole, the dataset triggers the #selectionListChanged event.

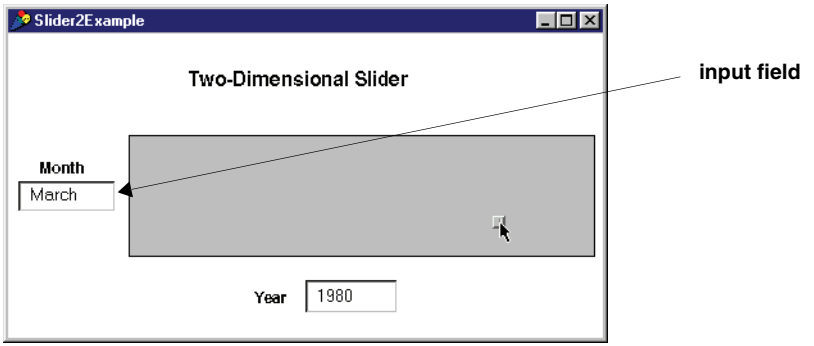
Input Fields

An input field is used for both entering and displaying data. You can also use a field in read-only mode when you just want to display data.

When a field has a short list of valid entries, consider using a menu button or a combo box instead.

A field uses a value model to manage its data (see Chapter 4, “Adapting Domain Models to Widgets”). When the field accepts input from a user, it sends this data to the value model for storage; when the field needs to update its display, it asks its value model for the data to be displayed.

Creating an Input Field



Online example: Slider2Example

1. From the Palette, add a field to the canvas.
2. On the **Basics** property page, in the field’s **Aspect** property, enter the name of the get method for the field’s value model.
3. On the **Details** page, select or assign values for the field’s properties.

   The size property takes an integer value for the maximum number of characters allowed.

   The following subsections describe other properties.
4. Apply the properties and install the canvas.
5. Use the **define** command to add the instance variable to the application model and define the aspect accessor method.
6. Define an initialize method for the aspect instance variable, or add initialization to the aspect accessor method, and evaluate the method to initialize the variable.

In this example, the variable is initialized to the desired month using an initialize method.

```
initialize
    month := (Date nameOfMonth: 1) asValue.
    year := 1900 asValue.

    dateRange := (0@1) asValue.
    dateRange onChangeSend: #changedDate to: self.
```

Restricting Input Type

You specify the type of input that a field is to accept by setting its **Type** property. This property converts the input string into an appropriate kind of object before sending it to the value model. If the conversion cannot be performed, the field flashes and continues to display the unaccepted string without storing it in its value model.

> **Note:** Ensure that the field is initialized with the appropriate type of data.

You can choose from the following data types. If none of these types are appropriate, you can add your own type testing.

Data Type	Description
String	Input is stored as a ByteString. (Default)
Symbol	Input is stored as a Symbol. Useful for applications that manipulate method selectors.
Text	Input is stored as an instance of Text, and supports formatting.
Number	Input is stored as an appropriate subclass of Number.
Password	Input is stored as a string, with an asterisk (*) displayed for each character the user enters. (The actual characters are sent to the field's value model.)
Date	Input is converted into an instance of Date.
Time	Input is converted into an instance of Time.
Timestamp	Input is converted into an instance of Timestamp.
FixedPoint(2)	Input is converted to a FixedPoint with two decimal places. Useful for applications that manipulate monetary amounts.
Boolean	Input is stored as an instance of Boolean. Accepts true and false only.

Data Type	Description
ByteArray	Input is stored as an instance of ByteArray.
Object	Input is evaluated as a Smalltalk expression, and the resulting object is stored as the field's value. The field redisplays this object using the object's printString method. Note that the VisualWorks compiler is required to run an application with Object as the data type for an input field. This is permitted by the license, and the compiler is not automatically removed by RuntimePackager.

Formatting Displayed Data

For some data types, the displayed string can be formatted in various ways. For example, numbers can be formatted as phone numbers, monetary units, percentages, and so on.

Predefined data type formats are provided by drop-down lists on the **Basics** properties page. The field's **Type** property setting determines the kinds of available formats, if any.

For a description of the format conventions, browse the class comments for the NumberPrintPolicy, TimestampPrintPolicy, and StringPrintPolicy classes.

Creating a Custom Format

Online example: FieldTypeExample

A TypeConverter enables a field to display a number in a special format, such as a monetary format. You define the format as a string that uses the same conventions as the predefined formats.

Edit the aspect accessor method to initialize the value to a TypeConverter, and specify the format string. This example shows how to create a format for a monetary amount.

```
price
    ^price isNil
        ifTrue: [price := (TypeConverter
            onNumberValue: 0 asValue
            format: '$###,###,###.##')]
              ifFalse: [price]
```

Validating Input

Frequently, only certain entries are valid for a particular field. For example, you might want to restrict input to a numeric range such as 0 to 999 or check for undesirable characters.

For a general description of the Validation properties, refer to "Validation Properties" in Chapter 3.

Data can be validated either a whole-field at a time, or character by character. The following subsections describe a few variations on these options.

Validating a Whole Entry

Online example: FieldValidInputExample

Most often you only need to validate an entire entry in a field. To do this, you need to specify and define the validation message which is sent when appropriate based on the user's actions.

1. On the **Validation** page of the GUI Painter Tool, enter the names of validation methods for one or more of the validation points.

   In the example, we use:

   - the **Change** property, to determine whether to accept input into the field's value model
   - the **Exit** property, to determine whether the field can give up focus

   Both send validateUsername:.

2. On the **Basics** property page, enter the aspect name (username).
3. Apply properties and install the canvas, and use **define** to add the aspect instance variable (username) and aspect method (username) to the application model. Initialize the instance variable with a value model.
4. Create the validation method (validateUsername:) entered in the validation fields.

Because the Change validation needs to validate the data before it is sent to the field's value holder, the validation method needs to get the entered value from the widget's controller. By naming the method so its selector ends in a colon, the widget sends the controller as an argument, which the method can then use.

In the validation method, send an editValue message to the field's Controller to obtain the user's entry. The entry must be obtained from the controller instead of the value model, because validation occurs before the entry has been passed to the value model.

The validation in the example consists of checking the entry's length.

If the entry is valid, the validation method returns true, allowing the field to pass the entry to the value model and, if requested, give up focus. If the entry is not valid, the method must return false. In the example, a warning is also returned, telling the user to correct the entry.

```
validateUsername: aController
    "Check the length of the entered username. Warn the user if the
    entered input is too long."

    | entry lengthLimit |
    lengthLimit := 6.
    entry := aController editValue.

    "If the username is too long, warn the user (and reject
    the input)."
    ^entry size <= lengthLimit
        ifTrue: [true]
        ifFalse: [Dialog warn: 'Please enter only ', lengthLimit
        printString ,
                ' characters.'.
            false]
```

Validating Individual Characters

Online example: FieldValidation1Example

In some cases, it is more useful to verify the user's entry at each keystroke. This approach requires more constant monitoring of the widget.

1. On the Basics property page, in the **ID** field, enter an identifier for the field (**codeField**). Apply the property and install the canvas.
2. Define a postBuildWith: instance method in the application model (FieldValidation1Example).

   In this method, the first task is to get the controller from the field. Next, it sends a keyboardHook: message to the controller. The argument to keyboardHook: is a two-argument block that takes the keyboard event and the controller as arguments.

   Inside the block, invoke the validation method (keyPress:). Alternatively, you can put the validation code directly inside the block.

```
postBuildWith: aBuilder
    | ctrlr |
    ctrlr := (self controllerAt: #codeField).
    ctrlr keyboardHook: [ :ev :c |
        self keyPress: ev].
```

3 Define the validation method (keyPress:).

This method takes the keyboard event as its argument, extracts a character, and validates it.

As the last step in the keyPress: method, return the event when you want to forward the keyboard event for normal processing. Return nil to bypass normal processing.

```
keyPress: ev
    "Validate the character."

    | ch ascii |
    ch := ev keyValue.

    "Allow tab and cr."
    ascii := ch asInteger.
    (ascii == 9 or: [ascii == 13])
        ifTrue: [^ev].

    ch isAlphaNumeric
        ifFalse: [
            Dialog warn: 'Please enter only letters and digits'.
            ^nil].
    ^ev
```

Modifying a Field's Pop-Up Menu

Online example: FieldMenuExample

By default, a field has a menu of text-editing commands. You can add or omit commands, override the action that is associated with a command, or disable the menu entirely.

A field's menu is usually oriented toward commands. Although you can arrange for a field's menu to contain a list of valid entries, this is properly the job of a menu button.

Adding a Command

1 In the field's **Menu** property, enter the name of the method that you will create to supply a custom menu (expandedMenu).

2 Define the menu-creating method (expandedMenu) in the application model, in a menu messages protocol.

```
expandedMenu
    "Add a command to the default text-editing menu."

    | mb |
    mb := MenuBuilder new.
    mb
        add: 'capitalize'->#capitalize;
        line;
        addDefaultTextMenu.

    ^mb menu
```

3 Define the method (capitalize) that is invoked by the newly added command. Put the method in the menu messages protocol.

```
capitalize
    "Capitalize the field's contents."

    self field1 value: (self field1 value
        collect: [ :ch | ch asUppercase]).
```

Overriding a Default Command

You can override a default command by building the menu from its parts. For the command that you want to override, provide the name of your overriding method as the value (#newAccept).

```
newAcceptMenu
    "Redefine the 'accept' command by invoking a local alternate."

    | mb |
    mb := MenuBuilder new.
    mb
        addFindReplaceUndo;
        line;
        addCopyCutPaste;
        line;
        add: 'accept'->#newAccept;
        add: 'cancel'->#cancel.
    ^mb menu
```

Then define the overriding method (newAccept).

```
newAccept
    Transcript show: self field3 value; cr.
```

Disabling a Field's Menu

To disable the menu, build a menu creation method that returns a block containing nil. When asked for its menu, the field will evaluate this block, and no menu is displayed.

```
noMenu
    ^[nil]
```

Connecting two Fields

Online example: FieldConnectionExample

When the value in a field depends on the value in another field, you can link them using the built-in dependency mechanism. The connection can be either one-way or two-way, depending on the needs of your application.

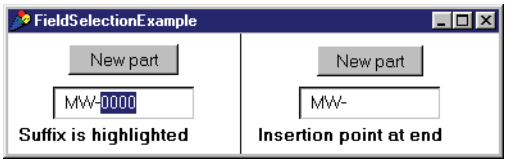
To define the dependency, you must register an interest between the fields. Create a postBuildWith: method in the application model, in an interface opening protocol, and register the interest in the field that originates updates. In the onChangeSend:to: method, name a method to be invoked when that field is changed (changedA).

```
postBuildWith: aBuilder
    self a onChangeSend: #changedA to: self.
```

Then define the change method (changedA) in the application model in a change messages protocol. That method updates the dependent field's model.

```
changedA
    self aSquared value: (self a value raisedTo: 2).
```

Controlling the Insertion Point



You can control the position of the insertion point in a field programmatically. For example, the data in a field might have a prefix that rarely changes—you could highlight the suffix for convenient editing. In that case, the "insertion point" is actually a portion of the field's text, which will be replaced by the user's entry.

When the suffix has yet to be filled in, you can simply position the insertion point at the end of the prefix.

Highlighting a Portion of a Field

Online example: FieldSelectionExample

1. In a method in the application model, ask the field's wrapper to takeKeyboardFocus.
2. Tell the field's controller the indices (character positions) of the substring that is to be highlighted.

```
addPart
    "Put a template in the partID field, then highlight the suffix."

    | wrapper |
    self partID value: 'MW-0000'.

    wrapper := self wrapperAt: #part1.
    wrapper takeKeyboardFocus.
    wrapper widget controller selectFrom: 4 to: 7.
```

Positioning the Insertion Point

1. In a method in the application model, ask the field's wrapper to takeKeyboardFocus.
2. Tell the field's controller the character position at which to place the insertion point.

```
addPart2
    "Put a template in the partID2 field, then position the insertion
    point."

    | wrapper |
    self partID2 value: 'MW-'.

    wrapper := self wrapperAt: #part2.
    wrapper takeKeyboardFocus.
    wrapper widget controller selectAt: 4.
```

Input Field Events

An Input Field triggers events when its value is changing, when it gains and looses focus, when it gets clicks from the mouse, and when it's view scrolls to the left or right.

#changing

When the value of an input field is about to be accepted after directly editing the value and exiting the input field, the input field triggers the #changing event.

#changed

After the value of the input field has been accepted, the input field triggers the #changed event.

#clicked

When a input field is clicked on with the <Select> mouse button, the input field triggers the #clicked event.

#rightClicked

When a input field is right clicked on with the <Operate> mouse button, the input field triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the input field.

#doubleClicked

When a input field is double clicked on with the <Select> mouse button, the input field will trigger the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus

When a input field receives focus, either by being tabbed to or by clicking on by the mouse, the input field triggers the #gettingFocus event.

#losingFocus

When a input field loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the input field triggers the #losingFocus event.

#tabbed

When a input field has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the input field triggers the #tabbed event.

#backTabbed

When a input field has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order, the input field triggers the #backTabbed event.

#scrollLeft

If while editing, navigating with the keyboard, or selecting text with the keyboard or mouse in an input field, the view scrolls to the left, the input field triggers the #scrollLeft event.

#scrollRight

If while editing, navigating with the keyboard or selecting text with the keyboard or mouse in an input field, the view scrolls to the right, the input field triggers the #scrollRight event.

Hierarchical Lists

> **Note:** The Hierarchical List widget will be deprecated in a future release, in favor of the Tree View widget. Unless you need functionality in the Hierarchical List that is not currently provided by Tree View, please use Tree View. The missing functionality will be added to Tree View before Hierarchical List is deprecated.

A Hierarchical List widget is similar to a List widget, but its primary component is an IndentedTreeSelectionInList. This object holds a tree object representing the hierarchy.

For cases where the hierarchical list is displaying a natural hierarchy, such as the class hierarchy, IndentedTreeSelectionInList provides instance creation methods to create the tree. Otherwise, create the tree using instances of AssociationTreeWithParent.

Retrieving the index and contents of a list item uses the same methods as for a List widget's SelectionInList.

Adding a List

1. Use a Palette to add a Hierarchical List widget to the canvas. Leave the list selected.
2. On the **Basics** property page, fill in the list's **Aspect** property with the name of the method that will return an instance of IndentedTreeSelectionInList.
3. **Apply** the change, install the canvas, and use **define** to add an instance variable and aspect accessor method to the application model. This instance variable will hold the SelectionInList.
4. Write an initialize method to initialize the IndentedTreeSelectionInList.

Create a List for a Natural Tree

The IndentedTreeSelectionInList class has a few instance creation methods that make it easy to create a tree out of an object with a natural hierarchy. For instance, the class method #listObjectHierarchy:childAccessor:childNameAccessor: can create an appropriate list object as follows:

```
initialize
    theList := IndentedTreeSelectionInList
                listObjectHierarchy: Object
                childAccessor: #allSubclasses
                childNameAccessor: #name.
    ^self
```

For other instance creation convenience methods, browse the instance creation protocol.

Building a Tree

You can build a hierarchy of arbitrary objects using instances of AssociationTree or AssociationTreeWithParent. Then use the tree as the source of the hierarchy for the Hierarchical List widget.

This version of the initialize method creates a hierarchy:

```
initialize
    tree := (AssociationTreeWithParent key: 'first' value: #first).
    tree addChild:
        ((AssociationTreeWithParent
            key: 'second'
            value: #second)
            addChild: (AssociationTreeWithParent
                key: 'third'
                value: #third);
            addChild: (AssociationTreeWithParent
                key: 'fourth'
                value: #fourth));

        addChild: ((AssociationTreeWithParent
            key: 'fifth'
            value: #fifth)
            addChild: (AssociationTreeWithParent
                key: 'sixth'
                value: #sixth);
            addChild: (AssociationTreeWithParent
                key: 'seventh'
                value: #seventh)).

    myList := IndentedTreeSelectionInList
                listObjectHierarchy: tree
                childAccessor: #children
                childNameAccessor: #displayString.

    ^self
```

Hierarchical List Events

A Hierarchical List triggers events when a selection is changing, when the underlying list itself is changing, when it gains and looses focus, when it gets clicks from the mouse, when a node is expanded or collapsed, and when it's view scrolls.

#clicked

When a hierarchical list is clicked on with the <Select> mouse button, the hierarchical list triggers the #clicked event.

#rightClicked

When a hierarchical list is right clicked on with the <Operate> mouse button, the hierarchical list triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the hierarchical list.

#doubleClicked

When a hierarchical list is double clicked on with the <Select> mouse button, the hierarchical list triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus

When a hierarchical list receives focus, either by being tabbed to or by clicking on by the mouse, the hierarchical list triggers the #gettingFocus event.

#losingFocus

When a hierarchical list loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the hierarchical list triggers the #losingFocus event.

#tabbed

When a hierarchical list has focus, and the Tab key is pressed to move the focus to the next widget in the tab order, the hierarchical list triggers the #tabbed event.

#backTabbed

When a hierarchical list has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order, the hierarchical list triggers the #backTabbed event.

#scrollLeft

If while navigating with the keyboard or manipulating a horizontal scroll bar in a hierarchical list, the hierarchical list scrolls to the left, the hierarchical list triggers the #scrollLeft event.

#scrollRight

If while navigating with the keyboard or manipulating a horizontal scroll bar in a hierarchical list, the hierarchical list scrolls to the right, the hierarchical list triggers the #scrollRight event.

#scrollUp

If while navigating with the keyboard or manipulating a vertical scroll bar in a hierarchical list, the hierarchical list scrolls up, the hierarchical list triggers the #scrollUp event.

#scrollDown

If while navigating with the keyboard or manipulating a vertical scroll bar in a hierarchical list, the hierarchical list scrolls down, the hierarchical list triggers the #scrollDown event.

#selectionChanging

When an item in a hierarchical list is selected or unselected, before the change is applied, the hierarchical list triggers the #selectionChanging event.

#selectionChanged

After an item in a hierarchical list is selected or unselected, the hierarchical list triggers the #selectionChanged event.

#selectionListChanged

Whenever the list underlying a hierarchical list is manipulated, either by changing a value, reordering, adding or removing items, or changing the list as a whole, the hierarchical list triggers the #selectionListChanged event.

#itemExpanded

When an item in the hierarchical list is visually toggled from its collapsed view to its expanded view, the hierarchical list triggers the #itemExpanded event.

#itemCollapsed

When an item in the hierarchical list is visually toggled from its expanded view to its collapsed view, the hierarchical list triggers the #itemCollapsed event.

Labels

Online example: LogoExample

A label is used as a title or a description for another widget or group of widgets, such as a field. Since the text of a label can be changed while the application is running, a label can also be used for read-only display.

A label accommodates only a single line of text. For a multiline label, use a separate label for each line or use a read-only text widget.

Creating a Textual Label

To add a text label, select the Label widget and place it on the canvas.

On the **Basics** properties page, enter the text in the label's **Label** property field. Then **Apply** the properties and **Install** the canvas.

The label size automatically adjusts to accommodate your text.

Creating a Graphic Label

Use a graphic label when you want to add a pictorial element to an interface. The graphic can be changed while the application is running, so you can also use a graphic label to represent a changing aspect of the model pictorially.

A graphic label is passive. If you need the graphic to respond to a mouse click, use a graphic button instead.

For a large graphic that requires scroll bars, insert the graphic in a view holder instead.

To add a graphic label:

1. Use a Palette to place a label widget on the canvas
2. On the **Basics** property page, in the label's **Label** property, enter the name of the method that will supply the graphic image (in the example, **logo**).

   The method must be a class method, and by convention is placed in the resources protocol.
3. Turn on the **Label is Image** property.

   The **Supplied by Application** property is turned on as well, because the graphic will be supplied by the method named in step 2.
4. Apply the properties and install the canvas.

You can use the Image Editor to create the graphic image and install it in the application model, using the method name from step 2. You can also draw the image using graphics classes in VisualWorks, and create the class method to return it in the application model.

Making a Label Mnemonic

Adding a mnemonic to a label causes the cursor to jump to the selected widget when <Alt><key> is pressed.

The <key> is specified by including an ampersand (&) in the label string before the mnemonic letter. For example, if the label is **Label**, you could make L the mnemonic by changing the label string to **'&Label'** in the label's **String:** field. The label will show with the L underscored.

To associate the mnemonic with a widget, enter the widget ID in the label's **Details** page, in the **Mnemonic:** field.

Supplying the Label at Run Time

An application can change the content of a label while it is running. This is useful for using a label as a read-only display field.

Because the label changes size dynamically, be careful selecting text to avoid overlapping neighboring widgets.

To change the label, get the widget from the application model's builder. Then, set the label by sending a labelString: message to it with a string argument, or by sending a label: message to it with a composed text or graphic as the argument.

The following example updates a label before the canvas is opened, but you can change the label string at any time after the interface has been built.

```
postBuildWith: aBuilder
    "Update the slogan's text, and make the company name bold and red."

    | slogan txt emph label |

    "Insert the years-in-business into the slogan."
    slogan := 'Serving Shrimps For '
        , (Date today year - 1869) printString, ' Years'.
    ( self wrapperAt: #slogan ) labelString: slogan.

    "Make the company name bold and red."
    txt := 'Many Hands Shrimppickers' asText
    emph := Array
        with: #bold
        with: #color->ColorValue red.
    txt emphasizeFrom: 1 to: 10 with: emph.
    label := Label
        with: txt
        attributes: (TextAttributes styleNamed: #large).

    (self wrapperAt: #textLogo) label: label.
```

Building a Registry of Labels

When you plan to use a label, such as a company name or logo, in multiple interfaces, you can store it in a central registry. The system will look for the label in the registry when it does not find the usual resource method. Two system registries are available, one for graphics and the other for strings.

Registering a label is usually done in a class-initialization method, so the registration will occur whenever the class loaded into an image.

Note: Use registries sparingly, especially for graphic images, because each entry occupies memory until it is explicitly removed. For a non-registry alternative, consider using a preBuildWith: message in your application to assign the label to its builder as in

```
preBuildWith: aBuilder
    aBuilder labelAt: #tm put: '(TM)'.
```

To register a graphic image, send visualAt:put: to ApplicationModel. To register a string label, send labelAt:put: to ApplicationModel. The first argument is the name of the label, as defined in the **Label** property of the widget. The second argument is the graphic or string.

```
initialize
    "LogoExample initialize"

    "Register the graphic image for the trademark symbol."
    ApplicationModel
        visualAt: #trademark
        put: self trademark.

    "Register the textual version of the trademark symbol."
    ApplicationModel
        labelAt: #tm
        put: '(TM)'.
```

Execute the initialization method to make the label available.

To remove a label from the registry, get the appropriate registry by sending a visuals message to the ApplicationModel class for the graphics registry, or a labels message for the string registry. Then send the removeKey:ifAbsent: to the registry specifying the name of the label as the first argument. The second argument is a block specifying the action to be take if the label is not found, and can be empty.

```
"Visual registry"
| registry |
registry := ApplicationModel visuals.
registry removeKey: #trademark ifAbsent: [].

"Labels registry"
registry := ApplicationModel labels.
registry removeKey: #tm ifAbsent: [].
```

Label Events

Labels and Group Boxes trigger events only when their labels are changing.

#labelChanging
When a label or group box's label is about to change, the widget triggers the #labelChanging event.

#labelChanged
After a label or group box's label has changed, the widget triggers the #labelChanged event.

Lists

Online example: List1Example

A list widget is useful for displaying a collection of objects or as an input device, allowing the user to select one or more elements in the list.

A list widget depends on two value models, one to hold the collection of objects to be displayed, and the other to hold the index of the current selection. SelectionInList and MultiSelectionInList are the preferred models for a List widget, and contain both of the required value holders.

The List collection is particularly well suited for use with the List widget, rather than other sequenced collections. It is extensible, and propagates change messages to its dependents.

The elements in the collection can be any objects, provided that they can display themselves textually.

Adding a List

1. Use a Palette to add a list widget to the canvas. Leave the list selected.
2. On the **Basics** property page, fill in the list's **Aspect** property with the name of the method that will return an instance of SelectionInList (or MultiSeletionInList).
3. Use **define** to add an instance variable and aspect accessor method to the application model. This instance variable will hold the SelectionInList.
4. Write an initialize method to initialize the instance variable. You initialize the variable with an instance of SelectionInList that is itself initialized with a list of Smalltalk class names.

```
initialize
    super initialize.
    classes := SelectionInList with: Smalltalk classNames.
    classes selectionIndexHolder onChangeSend: #changedClass
        to: self.

    methodNames := MultiSelectionInList new.
    instances := SelectionInList new.
```

Changing the List of Elements

The contents of a list often change frequently. Changing the list is accomplished by giving the SelectionInList a new collection of elements. Note that this is not the same as installing an entirely new SelectionInList, which would break the link with the list widget.

In List1Example, both the Selectors view and the Instances view change whenever the selection in the Classes view is changed. The changedClass method is responsible for updating the list, which it does by getting the SelectionInList from the application model and sending a list: message to it, with the new List as the argument.

```
changedClass
    | cls |
    self classes selection isNil

        "No class is selected -- empty the selector list."
        ifTrue: [
            self methodNames list: List new.
            self instances list: List new]

        "A class is selected"
        ifFalse: [
            cls := Smalltalk at: self classes selection.

            "Update the selectors list."
            self methodNames list: cls selectors asList.

            "Update the instances list."
            self instances list: cls allInstances].
```

To set both a list and the current selection, send a setList:selecting: instead. This is illustrated in the SetListAndSelectionExample setListAndSelection method:

```
setListAndSelection

    self targetList setList: self removeSelect list copy
        selecting: self selectionList selection
```

The first argument is a new list, as for the list: method above. The second argument is a selected item. Both are set at once to avoid repositioning the target list, unless the selected item is also removed.

In the case of updating a list only, the refreshList: message takes the modified list, keeping the current selection as the selected item. The effect is to update the list without changing the selection display. The use of this method is illustrated in a few methods in RefreshSelectListExample, such as in the removeSelections method:

```
removeSelections
    | list |
    list := self removeSelect list.
    undoList add:(self removeSelect selectionIndexes asSortedCollection
        asArray collect:[:index| index -> (list at: index)]).
    redoList := OrderedCollection new.
    list removeAll: self removeSelect selections.
    self targetList refreshList: list copy
```

Editing the List of Elements

Instead of exchanging lists, as in the previous example, a list is often simply modified, by adding or deleting elements. In most cases, the current selection and list positioning should remain unchanged.

Enabling Multiple Selections

Sometimes it is appropriate for the user to select more than one item in a list as targets for an action. A list allows multiple selections when its **Multi Select** property is turned on.

A second property, **Use Modifier Keys For Multi Select**, determines how selections are to be made. When this property is turned on (the default), the user:

- Clicks the <Select> mouse button to select a single item on the list
- <Shift>-clicks to select additional contiguous items
- <Control>-clicks to select additional noncontiguous items

Both properties are typically turned on or off together.

1 On the **Details** property page, check List widget's **Multi Select** property, and make sure the **Use Modifier Keys For Multi Select** property is also checked. **Apply** properties and install the canvas.

2 In the application model's initialize method, initialize the list widget's aspect variable to hold a MultiSelectionInList (instead of a SelectionInList).

```
initialize
    super initialize.
    classes := SelectionInList with: Smalltalk classNames.
    classes selectionIndexHolder onChangeSend: #changedClass
            to: self.

    methodNames := MultiSelectionInList new.
    instances := SelectionInList new.
```

Getting a Selection Contents

When a list widget serves as an input device, your application needs to be able to find out which object is selected. You can ask a SelectionInList for the selected object or for the index of the selected object in the list. You can also set the selection programmatically.

For a multiselect list, there may be multiple selections or selection indexes, so your application model must be prepared to handle a collection of objects rather than a single selection or index.

When nothing is selected, a SelectionInList returns a nil object as the selection and zero as the index; a MultiSelectionInList returns an empty collection for either the selections or the indexes.

In the method that needs to know the current selection in the list, get the SelectionInList from the application model and send a selection message to it. To get the index only, send selectionIndex. For a MultiSelectionInList, use a selections or selectionIndexes message.

```
changedClass
    | cls |
    self classes selection isNil

        "No class is selected -- empty the selector list."
        ifTrue: [
            self methodNames list: List new.
            self instances list: List new]

        "A class is selected"
        ifFalse: [
            cls := Smalltalk at: self classes selection.

            "Update the selectors list."
            self methodNames list: cls selectors asSortedCollection.

            "Update the instances list."
            self instances list: cls allInstances].
```

Setting a Selection

It is sometimes useful for a program to set the selection.

To set a list selection, in the method that is to change the selection, get the SelectionInList from the application model and send it a selectionIndex: message with the desired index number as the argument. Alternatively, send a selection: message with the desired object itself as the argument.

```
postOpenWith: aBuilder

    super postOpenWith: aBuilder.

    "Uncomment the line below to auto-select the first class."
    self classes selectionIndex: 1.

    "Uncomment the lines below to auto-select the last class."
    "self classes selection: self classes list last.
    (self controllerAt: #classes) cursorPointWithScrolling."

    "In the classes list, use boxed highlighting instead of
    reverse-video."
    (self widgetAt: #classes) strokedSelection.
```

Note that, for a MultiSelectionInList, send selectionIndexes: or selections:, supplying as argument a collection of indexes or a collection of objects in the list.

To select all objects in a multiple-selection list, get the MultiSelectionInList from the application model and send a selectAll message to it.

```
selectAll
    self methodNames selectAll.
```

Similarly, to clear all selections, get the MultiSelectionInList from the application model and send a clearAll message to it.

```
clearAll
    self methodNames clearAll.
```

Connecting Two Lists

A list widget frequently interacts with another list, for example if one list contains items “in” a selection and the other list contains items that are “out.”

To connect two lists, you register an interest between them. In the application model’s initialize method, arrange for a change message to be sent to the application model whenever the selection is changed in the first list.

```
initialize
    super initialize.
    classes := SelectionInList with: Smalltalk classNames.
    classes selectionIndexHolder
        onChangeSend: #changedClass to: self.

    methodNames := MultiSelectionInList new.
    instances := SelectionInList new.
```

Then define the change method in the application model. This method tests whether anything is selected in the first list (classes) and then updates the second list (methodNames) appropriately.

```
changedClass
    | cls |
    self classes selection isNil

        "No class is selected -- empty the selector list."
        ifTrue: [
            self methodNames list: List new.
            self instances list: List new]

        "A class is selected"
        ifFalse: [
            cls := Smalltalk at: self classes selection.
            "Update the selectors list."
            self methodNames list: cls selectors asSortedCollection.
            "Update the instances list."
            self instances list: cls allInstances].
```

List Events

A List Box triggers events when a selection is changing, when the underlying list itself is changing, when it gains and looses focus, when it gets clicks from the mouse, and when it's view scrolls.

#clicked
When a list box is clicked on with the <Select> mouse button, the list box triggers the #clicked event.

#rightClicked
When a list box is right clicked on with the <Operate> mouse button, the list box triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the list box.

#doubleClicked
When a list box is double clicked on with the <Select> mouse button, the list box triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus
When a list box receives focus, either by being tabbed to or by clicking on by the mouse, the list box trigger the #gettingFocus event.

#losingFocus
When a list box loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the list box triggers the #losingFocus event.

#tabbed
When a list box has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the list box triggers the #tabbed event.

#backTabbed
When a list box has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order, the list box triggers the #backTabbed event.

#scrollLeft
If while navigating with the keyboard or manipulating a horizontal scroll bar in a list box, the list box scrolls to the left, the list box triggers the #scrollLeft event.

#scrollRight
If while navigating with the keyboard or manipulating a horizontal scroll bar in a list box, the list box scrolls to the right, the list box triggers the #scrollRight event.

#scrollUp

If while navigating with the keyboard or manipulating a vertical scroll bar in a list box, the list box scrolls up, the list box triggers the #scrollUp event.

#scrollDown

If while navigating with the keyboard or manipulating a vertical scroll bar in a list box, the list box scrolls down, the list box triggers the #scrollDown event.

#selectionChanging

When an item in a list box is selected or unselected, or in the case of a multiple select list box, the selections change, before the change is applied, the list box triggers the #selectionChanging event.

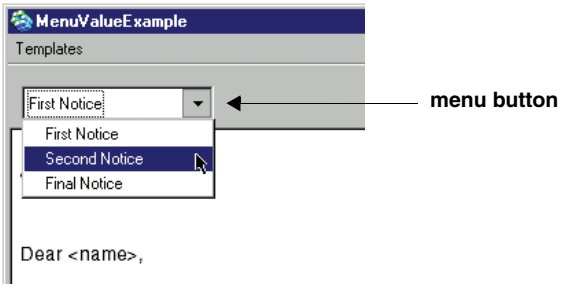
#selectionChanged

After an item in a list box is selected or unselected, or in the case of a multiple select list box, the selections have changed, the list box triggers the #selectionChanged event.

#selectionListChanged

Whenever the list underlying a list box is manipulated, either by changing a value, reordering, adding or removing items, or changing the list as a whole, the list box triggers the #selectionListChanged event.

Menu Button



A menu button allows you to place a drop-down menu anywhere in the canvas. Also, its label typically changes to reflect the current selection. By default, selecting an item in a button menu places the value of the menu item in the button menu widget's aspect value holder.

Adding a Menu Button

Online example: MenuValueExample

1. Add a menu button widget to the canvas and open the Properties Tool.
2. In the menu button's **Menu** property, enter the name of the menu creation resource method (templatesMenuForMenuButton).
3. In the button's **Aspect** property, enter a name for the aspect (letter).

   Leave the button's **Label** property blank. You can enter a label, but it will always be displayed in the menu. Leaving the **Label** blank displays the current selection on the menu button while the application is running. The initial (default) selection is set by initializing the aspect variable, typically in the initialize method.
4. Use the **define** command to add the aspect instance variable and accessor method (letter) to the application model.
5. Create or edit the initialize method to initialize the aspect variable to a value holder.

   Initializing the aspect variable also sets the initial item displayed on the menu button. This example also illustrates setting a check mark on the currently selected item.

```
initialize
    letter := self class firstNotice asValue.
    letter onChangeSend: #setCheckMark to: self.
```

6 Create the menu resource method that you named in step 2, and any action methods that are invoked by the menu items.

Adding a Menu Button with a Menu of Commands

Online example: MenuCommandExample

Adding a Button Menu that executes a command, such as an action block, differs from a Button Menu that stores a value in the initialization of the value holder. The value holder must be configured to perform its value whenever it is changed. This is typically done in an initialize method:

```
initialize
    files := SelectionInList new.
    files selectionIndexHolder onChangeSend: #configureMenu
        to: self.

    action := nil asValue.
    action onChangeSend: #performAction to: self.
```

You also must define a method that performs the currently selected action:

```
performAction
    self perform: self action value.
```

Menu Button Events

A Menu Button triggers events when its value is changing, when it gains and looses focus, when it gets clicks from the mouse, and when its list is exposed and closed.

#changing
When a selection in the list of a menu button is made, but before the value is accepted, the menu button triggers the #changing event.

#changed
After a selection in the list of a menu button is made and the value is accepted, the menu button triggers the #changed event.

#clicked
When a menu button is clicked on with the <Select> mouse button, the menu button triggers the #clicked event.

#gettingFocus
When a menu button receives focus, either by being tabbed to or by clicking on by the mouse, the menu button triggers the #gettingFocus event.

#losingFocus

When a menu button loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the menu button triggers the #losingFocus event.

#tabbed

When a menu button has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the menu button triggers the #tabbed event.

#backTabbed

When a menu button has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order, the menu button triggers the #backTabbed event.

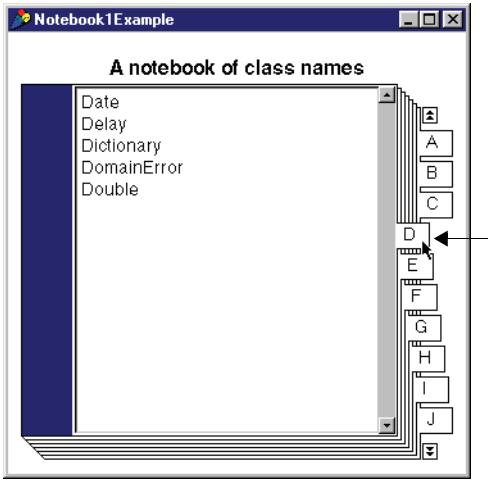
#listExposed

When a menu button has its selection list exposed, either by clicking on the menu button or using the down arrow key to open the list, the menu button triggers the #listExposed event.

#listClosed

When a menu button has its selection list closed, either by clicking on another widget on the canvas, by clicking on the menu button when the list is already exposed, or by selecting a new value in an exposed list, the menu button triggers the #listClosed event.

Notebook



A notebook is a powerful navigational widget. At its simplest, it provides a list in the form of index tabs. When the user selects an index tab, the effect is the same as selecting an item in a conventional list. A notebook can be used in many of the same situations in which a list or a menu might be used, but its richer set of capabilities (such as minor keys) extend its range of uses.

A notebook also contains a subcanvas which can be used to display a different interface for each index tab or, as in this simple example, the same interface.

In Notebook1Example, the subcanvas contains a list widget, and the list is changed each time an index tab is selected.

Creating a Notebook

Online example: Notebook1Example

1 Add a notebook widget to the canvas.

2 On the **Basics** page, enter the:

- **Major** property aspect name, the aspect that returns a SelectionInList containing major index tab labels
- **Minor** property aspect name, the aspect that returns a SelectionInList containing minor index tab labels (optional)
- **ID** property, then identifying name for the notebook (pageHolder)

3 Apply the properties and install the canvas, and **define** the instance variables and accessing methods for the notebook's major and minor index labels.

4 Initialize the major and minor variable, either in the accessing method or in an initialize method (as in the example), with a SelectionInList containing either strings or associations.

5 Create one or more additional canvases for the interface to display inside the notebook.

Install this canvas in its own resource method (listSpec), but in the same application model as the notebook, and **define** any variables and methods needed by the subcanvas. (In the example, these are the classNames variable, the classNames method, and the initialize method.)

For a notebook that uses different interfaces for each page, you will create several of these canvases.

6 Edit the initialize method to send an onChangeSend:to: message to send a changed message (changedLetter) to the application model when the user selects an index tab.

```
initialize
    | letters |
    letters := #( ' A' ' B' ' C' ' D' ' E' ' F' ' G' ' H' ' I' ' J' ' K' ' L' ' M'
            ' N' ' O' ' P' ' Q' ' R' ' S' ' T' ' U' ' V' ' W' ' X' ' Y' ' Z' ).
    majorKeys := SelectionInList with: letters.
    majorKeys selectionIndexHolder
        onChangeSend: #changedLetter to: self.
    classNames := SelectionInList new.
```

7 Create the change message (changedLetter) to update the subcanvas.

```
changedLetter
    | chosenLetter list |
    chosenLetter := self majorKeys selection last.
    list := Smalltalk classNames
        select: [ :name | name first == chosenLetter].
    self classNames list: list.
```

8 Create a postOpenWith: method to install the subcanvas.

In this method, get the notebook from the application model's builder, using the notebook's **ID** (pageHolder). Then install the subcanvas by sending a client:spec: message to the notebook. The first argument is the subcanvas's application model (self), and the second argument is the subcanvas's spec method (listSpec).

```
postOpenWith: aBuilder
    (self widgetAt: #pageHolder)
        client: self
        spec: #listSpec.
    majorKeys selectionIndex: 1.
```

Setting the Starting Page

Online example: Notebook1Example

By default, a notebook opens on a blank page, but another page provides better visual clues to the user as to the nature of the notebook. You can select the opening page either by setting the selection indexes of the major and minor SelectionInLists, or by specifying the list element itself.

In an instance method (such as postOpenWith:), send a selectionIndex: message to the SelectionInList that holds the major keys. The argument is the index of the desired tab.

```
postOpenWith: aBuilder
    (self widgetAt: #pageHolder)
        client: self
        spec: #listSpec.
    majorKeys selectionIndex: 1.
```

If the notebook has minor keys, also set the selection index for that SelectionInList.

Alternatively, to set the page by specifying the list element, send a selection: message to the SelectionInList that holds the major keys and, if applicable, another such message to the minor list. The argument is the desired element in the list.

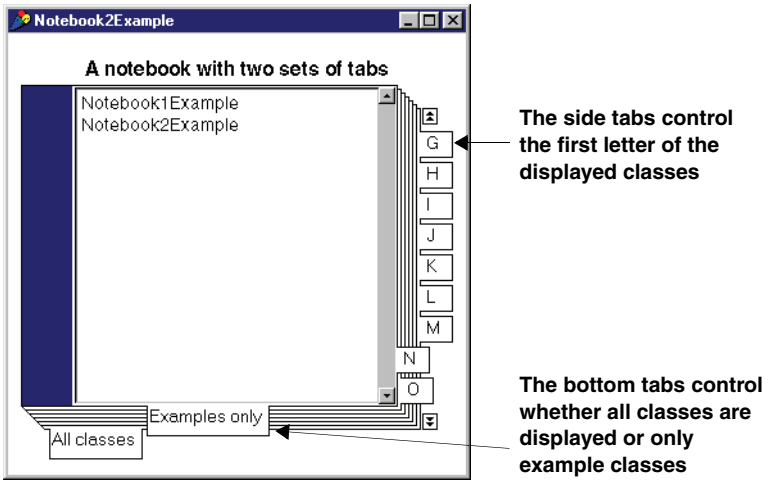
Getting the Selected Tab

When the user selects an index tab on a notebook widget, the selection changes in the underlying SelectionInList. Accessing that selection is a fundamental operation because the application model must know which tab is selected before it can take the appropriate action.

In a method in the application model, get the selected index tab's string or association by sending a selection message to the notebook's major SelectionInList. (In the example, the resulting string contains a leading space, so a last message is sent to get the index letter that follows the space).

```
changedLetter
    | chosenLetter list |
    chosenLetter := self majorKeys selection last.
    list := Smalltalk classNames
        select: [ :name | name first == chosenLetter].
    self classNames list: list.
```

Adding Secondary Tabs (Minor Keys)



Online example: Notebook2Example

In addition to the first set of index tabs, a second row of tabs can be added along another edge of the notebook. This second set of tabs is referred to as the *minor keys*. The minor keys can be used either to refine the subdivisions implied by the major keys or to filter the content of the notebook along a separate dimension.

In Notebook2Example, which lists the classes in the system, two minor keys are used to control whether the page shows all classes beginning with the selected letter or just the example classes.

1. On the **Basics** property page, fill in the notebook's **Minor** property with the name of the method that returns a SelectionInList containing the labels for the secondary tabs (in the example, minorKeys).
2. Use a System Browser or the canvas's **define** command to create the instance variable (minorKeys) and accessing method (minorKeys) for the notebook's list of index labels.

```
minorKeys
    ^minorKeys
```

3 Initialize the variable, either in the accessing method or in an initialize method (as in the example), with a SelectionInList containing either strings or associations (the example uses associations).

4 In the initialize method, use an onChangeSend:to: message to arrange for the notebook to send a message (changedPage) to the application model when the user selects a secondary tab. (In the example, both the major and minor tabs trigger the same message: changedPage.)

```
initialize
    | letters |
    letters := #( ' A' ' B' ' C' ' D' ' E' ' F' ' G' ' H' ' I' ' J' ' K' ' L' ' M'
            ' N' ' O' ' P' ' Q' ' R' ' S' ' T' ' U' ' V' ' W' ' X' ' Y' ' Z' ).
    majorKeys := SelectionInList with: letters.
    majorKeys selectionIndexHolder
        onChangeSend: #changedPage to: self.

    minorKeys := SelectionInList with: (Array
        with: 'All classes'-> #all
        with: 'Examples only' -> #examples).
    minorKeys selectionIndexHolder
        onChangeSend: #changedPage to: self.

    classNames := SelectionInList new.
```

5 Create the change message (changedPage) in which the subcanvas is updated based on the selected index tab. (In the example, the classNames list is updated with all class names or only example classes, based on the minor key.)

```
changedPage
    | chosenLetter list filter filteredList |

    "Major key."
    chosenLetter := self majorKeys selection last.
    list := Smalltalk classNames
        select: [ :name | name first == chosenLetter].
    list addAll: (Examples classNames select:
        [ :name | name first == chosenLetter]).
```

```
"Minor key."
filter := self minorKeys selection value.
filter == #all
    ifTrue: [filteredList := list]
    ifFalse: [filteredList := list
        select: [ :name | '*Example' match: name]].

self classNames list: filteredList.
```

Connecting Minor Tabs to Major Tabs

Online example: Notebook3Example

The major and minor keys of a notebook can be used to navigate through a two-tiered hierarchy of information. The minor keys can depend on the major keys so that when the user selects a major key, a different set of minor keys is presented.

1 In the application model's initialize method, set the two SelectionInLists to send onChangeSend:to: messages to notify the application model when their selections are changed.

```
initialize
    self initializeDepartments.
    self initializeEmployees.

    majorKeys := SelectionInList with: departments keys asArray.
    majorKeys selectionIndexHolder
        onChangeSend: #changedDepartment
        to: self.

    minorKeys := SelectionInList new.
    minorKeys selectionIndexHolder
        onChangeSend: #changedSubdepartment
        to: self.

    employeeList := SelectionInList new.
```

2 Create the change message (changedDepartment) for the major keys.

This method gets the selection from the major SelectionInList, and uses that selection for choosing the new labels for the minor tabs. The minor key selection is reset to display the first subpage.

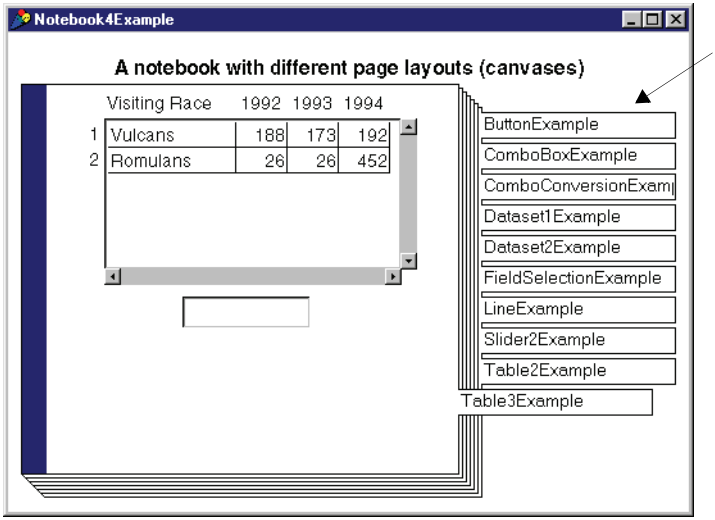
```
changedDepartment
    | subdepts sel |
    sel := self majorKeys selection.
    sel isNil ifTrue: [^self].

    "Display the appropriate subdepartments as minor keys."
    subdepts := self departments at: sel.
    self minorKeys list: subdepts.
    self minorKeys selectionIndex: 1.
```

3 Create the change message (changedSubdepartment) for the minor keys.

This method gets the minor selection and uses that selection for updating the canvas in the notebook.

```
changedSubdepartment
    "Display the appropriate employees in the list."

    | emps sel |
    sel := self minorKeys selection.
    sel isNil ifTrue: [^self].

    emps := self employees at: sel.
    self employeeList list: emps.
```

Changing the Page Layout (Subcanvas)



Online example: Notebook4Example

It is often useful for a notebook to present a different interface on each page. This is the basis of, for example, “wizards.”

In Notebook4Example, each index tab represents an example application. Selecting a tab causes a working instance of that application to be contained in the notebook.

1 In the application model’s initialize method, set the major keys SelectionInList to notify the application model when a tab is selected.

```
initialize
    | exampleClasses |
    exampleClasses := OrderedCollection new.
    exampleClasses := Examples keys select: [ :c |
        (('*Example' match: c)
                and: [(Smalltalk at: c) isVisualStartable])
                and: [('Notebook*' match: c) not]].
    majorKeys := SelectionInList
        with: exampleClasses asSortedCollection.

    majorKeys selectionIndexHolder
        onChangeSend: #changedExample to: self.
```

2 In the change method (changedExample), get the notebook widget and send it a client:spec: message to the notebook.

The first argument is the application model (in the example, exampleClass). The second argument is the spec for the desired canvas (in the example, each example class’s windowSpec is used).

```
changedExample
    | sel exampleClass |
    sel := self majorKeys selection.
    sel isNil ifTrue: [^self].

    exampleClass := Examples at: sel value.

    (self widgetAt: #pageHolder)
        client: exampleClass new
        spec: #windowSpec.
```

Notebook Events

A Notebook triggers events when a tab selection is changing, when one of its tabs gains or looses focus, and when it's tabs scroll.

#gettingFocus

When any of a notbook's tab groups (either horizontal or vertical) receives focus by being clicked on by the mouse, or tabbed into by the keyboard, when none of that group of tabs have had focus, the notebook triggers the #gettingFocus event.

#losingFocus

When any of a notebook's tabs groups (either horizontal or vertical) lose focus, by being clicked or tabbed away from, to anything except another of the tabs in the same group, the notebook triggers the #losingFocus event.

#tabbed

When a notebook tab group (either horizontal or vertical) has focus, and the and the Tab key is pressed to move the focus to the next widget in the tab order or another tab group on the notebook, the notebook triggers the #tabbed event.

#backTabbed

When a notebook tab group (either horizontal or vertical) has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order or another tab group on the notebook, the notebook triggers the #backTabbed event.

#scrollLeft

If all of the tabs in a horizontal tab group of a notebook are not visible, and the left double arrow icon is pressed to scroll additional horizontal tabs into view, then the notebook triggers the #scrollLeft event.

#scrollRight

If all tabs in a horizontal tab group of a notebook are not visible, and the right double arrow icon is pressed to scroll additional horizontal tabs into view, then the notebook triggers the #scrollRight event.

#scrollUp

If all tabs in a vertical tab group of a notebook are not visible, and the up double arrow icon is pressed to scroll additional vertical tabs into view, then the notebook trigger the #scrollUp event.

#scrollDown

If all tabs in a vertical tab group of a notebook are not visible, and the down double arrow icon is pressed to scroll additional vertical tabs into view, then the notebook triggers the #scrollDown event.

#horizontalTabChanging

When a new tab within a horizontal tab group is selected, either by selection using the mouse or by keyboard navigation, before the view changes to the newly selected tab the notebook triggers the #horizontalTabChanging event.

#horizontalTabChanged

After a new tab within a horizontal tab group is selected, either by selection using the mouse or by keyboard navigation, the notebook triggers the #horizontalTabChanged event.

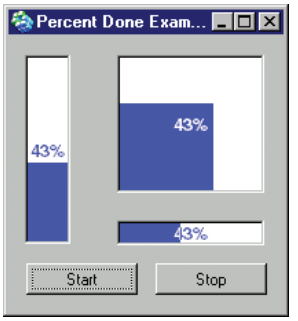
#verticalTabChanging

When a new tab within a vertical tab group is selected, either by selection using the mouse or by keyboard navigation, before the view changes to the newly selected tab the notebook triggers the #verticalTabChanging event.

#verticalTabChanged

After a new tab within a vertical tab group is selected, either by selection using the mouse or by keyboard navigation, the notebook triggers the #verticalTabChanged event.

Percent Done Bar



Also sometimes called a Progress Bar, this widget gives a visual indication of how far a process has advanced. It's commonly used for file copy procedures, but can be useful for any number of lengthy processes to give the user an indication of how much work has been done and how much more there is to go.

Basic Properties

Only the Basic properties are specific to this widget.

Aspect

The instance variable that determines the percentage. The value is a numeric value between 0 and 1.

Orientation

Horizontal indicates progress horizontally.

Vertical indicates progress vertically.

Both indicates progress both vertically and horizontally.

Area shows percentage in terms of area covered, rather than linearly.

Reverse reverses the horizontal and/or vertical direction.

Starting Point

The grid with five radio buttons sets the starting point: top, bottom, left, right, or center. If Both is selected for the orientation, then the starting point is either a corner or the center.

Adding a Percent Done Bar

1. Add a Percent Done Bar widget to the canvas and select it.
2. In the **Aspect** property, enter a name for the aspect.
3. Set the progress orientation: **Horizontal**, **Vertical**, or **Both**.

   Optionally, select **Reverse**, to reverse the direction of progress indication. Select Area to indicate progress as the percentage of area filled.
4. Select a starting point radio button.
5. **Apply** the properties and install the canvas. Use **define** to add the aspect instance variable and accessor method.

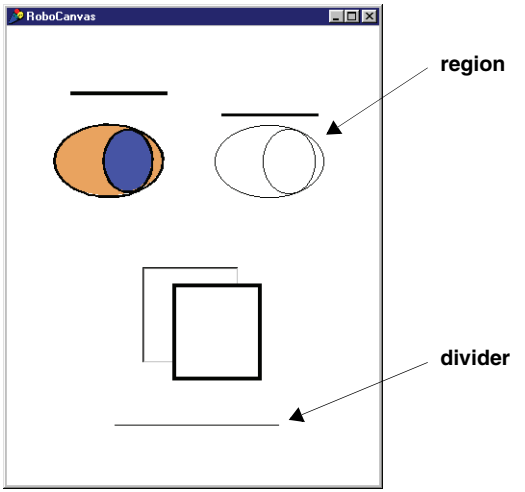
Percent Done Bar Events

A Percent Done Bar triggers events when its value has changed.

#changed

When a percent done bar is given a new percentage value, it triggers the #changed event. The percent done bar, unlike other widgets that trigger #changed events, does not have a matching a #changing event.

Lines, Boxes, and Ovals



Lines, group boxes, and regions (rectangles and ellipses) are used to provide visual separation between sets of interface components. Using appropriate dividers can greatly improve the usability of an interface.

Adding a Line to a Canvas

Online example: LineExample

1. Use a Palette to add a divider to the canvas.
2. On the **Basics** property page, select either **Horizontal** or **Vertical** for the line's orientation property. Apply the property.
3. Use the widget handles to size and position the divider.
4. Install the canvas.

Lines are one pixel thick. For a thicker line, use a rectangular region and set its **Thick** border property.

Adding a Group Box

When an interface begins to appear cluttered, the user of your application may have trouble understanding how the widgets relate to one another. As a visual aid, cluster the widgets in logical groups. Spacing is one way to group widgets; another way is to surround some groups with boxes.

A box can have a label embedded in its top border. Its line thickness is one pixel. For a thicker line, use a region as described below.

1. Use a Palette to add a box to the canvas. Leave the box selected.
2. Use the widget handles to size and position the box.
3. Fill in the **Label** property, if desired.
4. Choose the label's font.
5. Apply the properties and install the canvas.

A box line thickness is one pixel. For a thicker line, use a rectangular region and set its border property to **Thick**. Also use a region if you want to use color.

Making a Group Box Mnemonic

Adding a mnemonic to a Group Box label causes the cursor to jump to the selected widget when <Alt><key> is pressed.

The <key> is specified by including an ampersand (&) in the Group Box label string before the mnemonic letter. For example, if the label is **Label**, you could make L the mnemonic by changing the label string to **'&Label'** in the Group Box's **String:** field. The label will show with the L underscored.

To associate the mnemonic with a widget, enter the widget ID in the Group Box's **Details** page, in the **Mnemonic:** field.

Group Box Events

A Group Box triggers events only when its label is changing.

#labelChanging
When a label or group box's label is about to change, the widget triggers the #labelChanging event.

#labelChanged
After a label or group box's label has changed, the widget triggers the #labelChanged event.

Grouping Widgets with a Region

For visual variety, you can use a region. A region can be rectangular or elliptical, and supports colors.

1. Use a Palette to add a region to the canvas. Leave the region selected.
2. Set the region's **Rectangle** or **Ellipse** property, as desired, and apply the properties.
3. Use the widget handles to size and position the region.

4 If desired, use the **Color** property page to apply color to the foreground (border) and/or background (interior).

5 **Apply** the properties and install the canvas.

Region Events

A Region triggers events when one of its visible properties is changed programmatically.

#changing

When a region is about to change its size, inside color, border color or border size, before the change is applied the region triggers the #changing event.

#changed

After a region has had its size, inside color, border color or border size changed, the region triggers the #changed event.

Resizing Splitter

The Resizing Splitter widget allows users to resize widgets in the GUI. For example, in the GUI Painter Tool, a Resizing Splitter is between the hierarchical list of widgets and the tabbed notebook.

You can change the relative size of both by selecting and dragging the area between them. The cursor shape changes while it is passing over a Resizing Splitter, indicating that the border is movable.

Because this widget is really only intended for direct user manipulation, no programmatic interface is described in this section.

Typically, the Resizing Splitter will be one of the last widgets you add to the canvas, since you add it between other widgets whose size you want to allow the user to control.

Adding a Resizing Splitter

1. Select the Resizing Splitter and drop it on the canvas between the widgets you want to resize.
2. Initially the widget is proportioned horizontally, and the the **Horizontal** check box is marked, for separating widgets top to bottom. To separate widgets left to right, uncheck the **Horizontal** check box and resize the widget accordingly.
3. In the **Left/top widgets:** properties field, enter the IDs of the widgets to the left of or above the widget. Similarly, in the **Right/bottom widgets:** field, enter the IDs of the widgets to the right or below the Resizing Splitter.

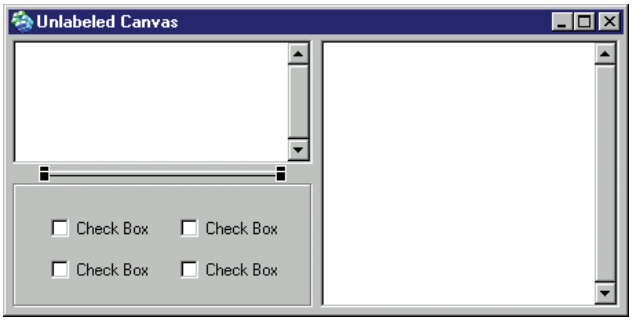
   Since there may be (frequently are) more than one widget on one side of a widget, these fields take space-separated widget ID names. Only widgets in the lists are resized as the Resizer Splitter is moved. IDs can be listed either simply as identifiers (**TreeView1 TextEditor2**) or as symbols (**#TreeView1 #TextEditor2**).
4. **Accept** and install the canvas.

Setting Widget Positioning

To make sure the widgets resize appropriately as the ResizIng Splitter is moved, you will need to set the widget positions carefully. Use the **Position** pages for the affected widgets' properties for precise control.

The Resizing Splitter can either have a border or be borderless. With a border it can double as a separator line, and so enhance the look of your GUI. Without a border it is invisible, and so simply serves as an area to grab for resizing. Even if you want it to be invisible in the final product, leaving it with a border can be useful while arranging your GUI.

For example, let's set position settings for a GUI with a Tree View in the top left and a group of Check Boxes on the lower left, separated by a bordered Resizing Splitter, and a Text Editor on the right separated by an unbordered Resizing Splitter. The GUI looks like this:



Set the positions as shown below (placement of the check boxes is left as an exercise for the reader):

	Position	Offset
TreeView (top left)		
Left (L)	0	3
Top (T)	0	3
Right (R)	0.5	-3
Bottom (B)	0.5	-7
TextEditor (right)		
Left (L)	0.5	3
Top (T)	0	3
Right (R)	1	-3
Bottom (B)	1	-3
GroupBox (bottom left)		
Left (L)	0	3
Top (T)	0	3

	Position	Offset
Right (R)	0.5	-3
Bottom (B)	0.5	-7
ResizingSplitter (horizontal)		
Left (L)	0	26
Top (T)	0.5	-2
Right (R)	0.5	-26
Bottom (B)	0.5	2
RisizingSplitter (vertical)		
Left (L)	0.5	-2
Top (T)	0	0
Right (R)	0.5	2
Bottom (B)	1	0

For clarity, the positions are all given as relative either to a line shared by a Resizing Splitter, or to an edge, but that is not necessary. The affected widgets are identified in the Resizing Splitter's **Left/top widgets:** and **Right/bottom widgets:** lists.

Resizing Splitter Events

A Resizing Splitter triggers events when it is clicked on and when its position on the canvas has changed.

#clicked

When a resizing splitter is clicked on by the mouse, without regard to if the splitter is moved, the resizing splitter triggers the #clicked event.

#moved

When a resizing splitter has finished being moved and then the mouse button has been released, the resizing splitter triggers the #moved event.

Sliders

A slider simulates the sliding switch that some electronic devices use for controlling volume, bass level, and other properties. A slider enables you as the designer of an application to define a specific range of legal values, and it enables the user to conveniently select a value within that range.

Adding a Slider

Online example: Slider1Example

1. Add a slider widget to the canvas.
2. On the **Basics** property page, in the slider's **Aspect** property, enter a name for the aspect (destination).
3. (Optional) Set the **Start**, **Stop**, and **Step** properties

   These properties set the range and increment for the slider. The default **Start** is 0 and **Stop** is 1. The default **Step** is **nil**, providing continuous motion.
4. Apply the properties and install the canvas, and use **define** to add the aspect instance variable and accessor method (destination).
5. Create or edit the initialize method to initialize the variable with a value holder and an initial value.

```
initialize
    "Destination"
    destination := Date today year asValue.

    "Current year"
    currentYear := Date today year asValue.

    "Trip meter"
    tripRange := RangeAdaptor
        on: currentYear
        stop: 4000
        grid: 1.
```

Connecting a Slider to a Field

Although a slider is both an input and an output device, it gives only a rough idea of the current value. A field is commonly used to display the precise value.

Slider1Example uses a field to display the destination year, because the slider covers such a large range (zero to 4000) that the user can only guess at its current value.

1 Add a field to the canvas.

2 In the field's **Aspect** property, enter the *same* aspect name as for the slider.

3 In the field's **Type** property, select **Number**.

4 Apply the properties and install the canvas.

Connecting a Slider to a Non-numeric Field

Online example: Slider2Example

By its nature, a slider always manipulates a numeric value. You can make it appear to manipulate a nonnumeric value, however, by using a field to display the transformed value. The example translates numeric values to months.

1 Add a field to the canvas.

2 In the field's **Aspect** property, enter a *different* method name than the slider's **Aspect** (month).

3 In the field's **Type** property, select the type that corresponds to the transformed value (the example uses a **String** type).

4 Apply the properties and install the canvas, and use **define** to create the aspect instance variable and accessing method.

5 In a method in the application model (typically initialize), initialize the field's variable.

6 Edit the initialize method so a change message (changedDate) is sent to the application model when the slider's value changes.

```
initialize
    month :=(Date nameOfMonth: 1) asValue.
    year := 1900 asValue.

    dateRange := (0@1) asValue.
    dateRange onChangeSend: #changedDate to: self.
```

7 Create the change method (changedDate) in the application model.

This method is responsible for changing the field's value based on the slider's new value.

```
changedDate
    "Convert the y-axis value to a month."
    | y x |
    y := self dateRange value y.
    y := (12 - (y * 12) asInteger) max: 1."(12 months)"
    self month value: (Date nameOfMonth: y).

    "Convert the x-axis value to a year."
    x := self dateRange value x.
    x := 1900 + (x * 100) asInteger."(100 years)"
    self year value: x.
```

Making a Slider Read-Only

Although it is normally an input device, a slider can be used purely as an output device. In Slider1Example, we use a read-only slider as a meter to display the progress of the user's time-traveling adventure.

1. On the **Basics** properties page, assign the slider's **ID** property with an identifying name (tripRange).
2. In a method in the application model (typically postBuildWith:), get the slider component from the builder and disable it.

```
postBuildWith: aBuilder
    "Disable the trip meter, making it read-only."
    (self wrapperAt: #tripRange) disable.
```

Changing the Slider Range

When the slider's range is unchanging, you can use the slider's **Start**, **Stop**, and **Step** properties to set the range and the step value. When the range or step varies, however, you need to adjust the range within the application.

A RangeAdaptor is a specialized value model that also keeps track of the range and step values. You can change those values by sending messages to the adaptor.

Online example: Slider1Example (the **Trip Meter** slider)

1. Edit the slider's initialization (typically in an initialize method) to initialize the slider's aspect variable with a RangeAdaptor by sending the instance creation message (on:start:stop:grid:).

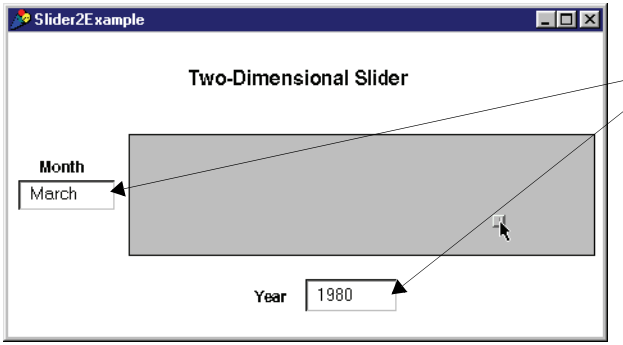
   The on: argument is a value holder containing the number that the slider manipulates. When a field is connected to the slider, as in the example, this argument is the field's aspect variable. The grid: argument is the step value.

```
initialize
    "Destination"
    destination := Date today year asValue.

    "Current year"
    currentYear := Date today year asValue.

    "Trip meter"
    tripRange := RangeAdaptor
        on: currentYear
        start: 0
        stop: 4000
        grid: 1.
```

2 Whenever the range or step must change, send a rangeStart:, rangeStop:, or grid: message to the adaptor. (In the example, this is done in the engage method.)

```
engage
    "Start the time trip."

    | startingYear destinationYear direction |
    startingYear := self currentYear value.
    destinationYear := self destination value.

    destinationYear == startingYear
        ifTrue: [^Dialog warn: 'Please select a new destination.'].

    "Set the endpoints on the trip meter."
    self tripRange
        rangeStart: startingYear;
        rangeStop: destinationYear;
        grid: 1.

    "Reset the meter to the starting position."
    currentYear value: startingYear.

    "Set up a step value for the loop that follows (-1 = backward in
     time)."
    destinationYear > startingYear
        ifTrue: [direction := 1]
        ifFalse: [direction := -1].

    "For each year of time travel, update the current year."
    startingYear to: destinationYear by: direction do: [ :yr |
        currentYear value: yr].
```

Making a Slider Two-Dimensional



You can make a slider manipulate a point in two dimensions and then use the *x*-axis and *y*-axis components of that point to control two separate parameters.

In Slider2Example, a two-dimensional slider is used to alter two fields simultaneously. The first field uses the *y* component to display one of the 12 months. The second field uses the *x* component to arrive at a year between 1900 and 2000.

Online example: Slider2Example

1 In an instance method (typically initialize), initialize the slider's variable to an instance of Point that is held by a value holder, and arrange for a change message (changedDate) to be sent to the application model when the slider's value changes.

```
initialize
    month := (Date nameOfMonth: 1) asValue.
    year := 1900 asValue.
    dateRange := (0@1) asValue.
    dateRange onChangeSend: #changedDate to: self
```

2 In a postBuildWith: method, get the slider from the application and ask it to beTwoDimensional.

```
postBuildWith: aBuilder

    (self widgetAt: #dateRange)
        beTwoDimensional;
        setMarkerLength: 10.
```

3 Create the change method (changedDate) in the application model.

This method splits the slider's value into its *x*-axis and *y*-axis components. Each component is a value between 0 and 1 and is

transformed as needed to produce a suitable value for the related field.

```
changedDate
    "Convert the y-axis value to a month."
    | y x |
    y := self dateRange value y.
    y := (12 - (y * 12) asInteger) max: 1."12 months"
    self month value: (Date nameOfMonth: y).

    "Convert the x-axis value to a year."
    x := self dateRange value x.
    x := 1900 + (x * 100) asInteger."(100 years)"
    self year value: x.
```

Slider Events

A Slider triggers events when its value has changed or folded in half and eaten whole (without regard to cheese). Unlike other widgets that interact with the mouse, a Slider never gets focus and thus does not trigger focus or mouse click events.

#changing

When a slider is about to change its value, the slider triggers the #changing event.

#changed

After a slider has changed its value, the slider triggers the #changed event.

Spin Buttons

A spin button allows a user to select a value from a determined range. The selected value is returned as a selectable object type and format.

A spin button model must represent a number, whenever its Type property is set to either **String**, **Text**, or **Symbol**. In these cases the model must respond to asNumber and not answer zero unless the model itself represents 0.

For example, model values for a spin button, listed by Type, might be:

Type	Example model value
String	'774.3'
Text	'12e-03' asText
Symbol	#'12'

The Low Value is the minimum setting the button will spin to, and the High Value is the maximum setting. The spin button range is unbounded unless these values are set.

The Interval is the amount each button press increments or decrements the spin button value. Any number may be entered for the Interval, but only positive numbers make sense. The default Interval value is 1.

For all spin button Types other than Date, Time, and Timestamp, the Low Value and High Value properties expect only numeric entries; no non-numeric symbols should appear in the entry.

If the Type property is set to Date, enter Low Value and High Value entries as dates, such as **12/20/1988** or **Sep 30 1990**. The Interval expects values as a number of days.

When Type is set to Time, format Low Value and High Value entries as times, such as **12:34:23 pm**. Do not set a High Value of **12:00:00 am**, unless you have the wrap around property set. The Interval entries are in units of seconds.

If Type is a Timestamp, enter Low and High Values in Timestamp format, such as **01/03/1999 0:00:00.000**. Interval entries are in units of milliseconds.

The table below summarizes the acceptable spin button property values for Date, Time, and Timestamp type.

Type	Low/High Value Entry Example	Interval Units
Date	12/20/1988, Sep 30 1990	Days
Time	12:34:23 pm, 18:00:00	Seconds
Timestamp	01/03/1999 0:00:00.000	Milliseconds

Adding a Spin Button

1. Add a spin button widget to the canvas.
2. In the **Aspect** property, enter a name for the aspect.
3. (Optional) Set the **Type**, **Format**, **Low Value**, **High Value**, and **Interval** properties

   These properties set the returned object type and format, and the range and increment for the presented values.
4. **Apply** the properties and install the canvas, and use **define** to add the aspect instance variable and accessor method.
5. Create or edit the initialize method to initialize the aspect variable with a value holder and an initial value.

Spin Button Events

A Spin Button triggers events when its value is changing, when it gains and looses focus, when it gets clicks from the mouse, when it spins up or down and it reaches the end of its value range.

#changing
When the value of the spin button is about to be accepted, either by spinning the button, or after directly editing the value and exiting the spin button, the spin button triggers the #changing event.

#changed
After the value of the spin button has been accepted, the spin button triggers the #changed event.

#clicked
When a spin button is clicked on with the <Select> mouse button, the spin button triggers the #clicked event.

#rightClicked

When a spin button is right clicked on with the <Operate> mouse button, the spin button triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the spin button.

#doubleClicked

When a spin button is double clicked on with the <Select> mouse button, the spin button will triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus

When a spin button receives focus, either by being tabbed to or by clicking on by the mouse, the spin button triggers the #gettingFocus event.

#losingFocus

When a spin button loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the spin button triggers the #losingFocus event.

#tabbed

When a spin button has focus, and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the spin button triggers the #tabbed event.

#backTabbed

When a spin button has focus, and the Back-Tab (Shift-Tab) key is pressed in request to have the focus move to the previous widget in the tab order, the spin button triggers the #backTabbed event.

#spinUp

When the up arrow button is pressed on a spin button, prior to changing the value of the spin button, the spin button triggers the #spinUp event. If the value of the spin button is changed, the #spinUp event is always immediately followed by #changing, #changed and #spunUp events. If the value of the spin button is not changed, the #spinUp event may be followed by a #bounceTop event, depending on the settings of the spin button.

#spunUp

After the up arrow button is pressed on a spin button, after changing the value of the spin button, the spin button triggers the #spunUp event. The #spunUp event is always immediately preceded by a #changed event.

#spinDown

When the down arrow button is pressed on a spin button, prior to changing the value of the spin button, the spin button triggers the #spinDown event. If the value of the spin button is changed, the #spinDown event is always immediately followed by #changing, #changed and #spunDown events. If the value of the spin button is not changed, the #spinDown event may be followed by a #bounceTop event, depending on the settings of the spin button.

#spunDown

After the down arrow button is pressed on a spin button, after changing the value of the spin button, the spin button triggers the #spunDown event. The #spunDown event is always immediately preceded by a #changed event.

#wrapAroundBottom

If a spin button has a low value set in its properties, and has the wrap around property set, then when the spin button is asked to spin below the lowest value, after the #spinDown, #changing, #changed and #spunDown events are triggered, the spin button triggers the #wrapAroundBottom event.

#wrapAroundTop

If a spin button has a high value set in its properties, and has the wrap around property set, then when the spin button is asked to spin above the highest value, after the #spinUp, #changing, #changed and #spunUp events are triggered, the spin button triggers the #wrapAroundTop event.

#bounceBottom

If a spin button has a low value set in its properties, and does not have the wrap around property set, then when the spin button is asked to spin below the lowest value, after the #spinDown event is triggered, the spin button triggers the #bounceBottom event. Because the value has not changed, the #changing, #changed and #spunDown events will not trigger.

#bounceTop

If a spin button has a high value set in its properties, and does not have the wrap around property set, then when the spin button is asked to spin above the highest value, after the #spinUp event is triggered, the spin button triggers the #bounceTop event. Because the value has not changed, the #changing, #changed and #spunUp events will not trigger.

Subcanvases

A subcanvas is a widget for displaying another canvas in the canvas you are building. It is an effective way to build an application interface that incorporates another application interface, or an entire application, into itself.

There are three primary uses for subcanvases:

Inheriting an application
Within a hierarchy of application models, you can inherit an interface and its value models.

Nesting an application
By placing an application in a subcanvas, you make its interface part of your own application's interface, and effectively incorporate its domain model into your application as well.

Reusing an interface only
You can incorporate another application's interface, but supply your own domain model for it.

All three of these methods promote various levels of reuse. Using subcanvases in these ways allow you to build a collection of reusable interfaces.

Inheriting a Subcanvas

Subcanvases provide a way to extend the inheritance mechanism in Smalltalk to building a user interface. By building an application model as a subclass of another application model, the subclass will inherit standard interface modules, value holders, and action methods.

For example, Subcanvas1Example is a subclass of List2Example, so it can reuse the List2Example interface, value holders, and actions.

A subclass need not reimplement anything that the parent class has implemented, but it *can* override an inherited action.

> **Note:** A limitation of the inheritance approach is that if you use the same inherited interface twice in a canvas, both will display the same thing. This is because both reference the same value holder.

Online example: List2Example (parent) and Subcanvas1Example

1 Create a new application model (Subcanvas1Example) as a subclass of the application model from which it is to inherit (List2Example).

2 Place a subcanvas widget on the inheriting canvas (the canvas for Subcanvas1Example).

3 In the subcanvas's **Canvas** property, enter the name of the inherited interface specification to be used by the subcanvas (listSpec).

This spec name must be unique within the inheritance chain. For example, you could not embed an inherited canvas named windowSpec in a local canvas named windowSpec.

The **Name** and **Class** properties are not needed for inheriting a canvas.

4 Apply the property and install the inheriting canvas in its application model (Subcanvas1Example).

Changing the Value of an Inherited Widget

The power of reuse is fully realized when you provide local values for the inherited widgets. For example, List2Example initializes its list to display a collection of color names. The inheriting application, Subcanvas1Example, provides its own collection, causing the reused list to display cursor names instead.

1 Create or edit the initialize method in the inheriting application model (Subcanvas1Example).

2 In the initialize method, invoke the inherited initialize method.

3 In the initialize method, use the inherited aspect message (selectionInList) to access the desired valued model. Then send an accessing message (in this case, list:) to the value model to install the desired value (cursorNames).

```
initialize
    "Install a different list (cursor names) than
    the inherited default (color names)."
    | cursorNames |
    super initialize.
    cursorNames := Cursor class organization
        listAtCategoryNamed: #constants.
    self selectionInList list: cursorNames.
```

Nesting One Application in Another

Using the subcanvas, you can embed (or nest) one application in another. This is useful, for example, for creating a set of reusable application modules that can be plugged into larger applications as needed. This approach can help avoid duplication of effort for generic modules and enforce interface design standards.

The embedded application supplies its own value models and action methods. Because the applications are not generally related by inheritance, you cannot override an embedded application's action methods.

You can embed the same application any number of times in the same canvas. For example, you could reuse List2Example four times in creating a System Browser's four list views.

Online example: List2Example embedded in Subcanvas2Example

1. Add a subcanvas widget in the reusing canvas.
2. Specify these properties for the subcanvas:

   Name, the name of the method that returns an instance of the embedded application (classNames).

   Class, the name of the application that you are embedding (List2Example).

   Canvas, the name of the interface specification for the embedded application (listSpec).
3. Apply the properties and install the reusing canvas in its application model (Subcanvas2Example).
4. Add an instance variable (classNames) in the reusing application model for holding onto the embedded application.
5. Add an accessor method for the instance variable:

```
classNames
    ^classNames
```

6. Create or edit the initialize method in the reusing application model, so it creates the embedded application and assigns it to the instance variable:

```
initialize
    "Reusing List2Example's interface only -- initialize the list
    holder."
    selectionInList := SelectionInList with: Smalltalk classNames.

    "Reusing List2Example application -- initialize the application
    instance."
    classNames := List2Example new.
    classNames list: Smalltalk classNames.
```

Setting a Value in an Embedded Widget

An embedded widget uses the value with which its host application initializes it.

1 In the initialize method, send a message (list:) to the embedded application, installing the desired value.

```
initialize
    "Reusing List2Example's interface only -- initialize the list
    holder."
    selectionInList := SelectionInList with: Smalltalk classNames.

    "Reusing List2Example application -- initialize the application
    instance."
    classNames := List2Example new.
    classNames list: Smalltalk classNames.
```

2 If necessary, as in the example, you may need to add a method (list:) to the embedded application model that allows an outside application to supply a new value.

```
list: aCollection
    "Install aCollection in the list. This message is provided so
    reuserscan install a list that is different than the default list
    (color names)."

    self selectionInList list: aCollection.
```

Reusing an Interface Only

You can use a subcanvas to embed one canvas inside another. This is similar to embedding an entire subapplication, except that the reusing application must supply all value models and methods for the interface.

Because you are reusing only the interface, you have to reimplement all of the supporting value holders and methods. You also have to supply actions for any buttons in the embedded interface.

Online example: Subcanvas2Example (which reuses List2Example's listSpec)

1 Add a subcanvas in the reusing canvas.

2 Specify these properties for the subcanvas:

Class, the name of the application that defines the interface to be embedded (List2Example).

Canvas, the name of the interface specification to be embedded (listSpec).

The **Name** property is not needed for reusing the interface only.

3 Apply the properties and install the reusing canvas in its application model (Subcanvas2Example).

4 Edit the reusing application model, adding instance variables and methods to support the embedded interface.

These instance variables and methods must have the same names as the corresponding variables and methods in the reused class. Modify values and action methods as desired.

Changing Interfaces at Run Time

Using a subcanvas makes it easy to change the UI in response to the current context of the application. In Subcanvas3Example, a subcanvas is used to hold either a text editor or a list view, depending on whether the user wants to see textual or listed material related to a selected class.

Online example: Subcanvas3Example

1 In the subcanvas's **Name** property, enter the name of the method that supplies the initial embedded application (embeddedApplication).

2 Apply the properties and install the reusing canvas.

3 Create the method (embeddedApplication) that you named in step 1.

4 Create a change method which:

a Creates and initializes an instance of the application model to swap (Editor2Example)

b Gets the specification for the new interface, by sending an interfaceSpecFor: message to the embedded application model's class (Editor2Example). The argument is the name of the interface specification (#windowSpec).

c Gets the subcanvas from the builder and send a client:spec: message to it. The first argument is the application you created in step 4a. The second argument is the spec object you obtained in step 4b.

```
showComment
    | selectedClass subcanvas spec application |
    selectedClass := Smalltalk at: self classNames selection.

    "Create the subapplication and initialize it."
    application := Editor2Example new.
    application text value: selectedClass comment.

    "Get the spec object for the embedded canvas."
    spec := Editor2Example interfaceSpecFor: #windowSpec.

    "Get the subcanvas and install the editing application."
    subcanvas := self widgetAt: #subcanvas.
    subcanvas client: application spec: spec.
```

Accessing an Embedded Widget

You may need to access a widget within an embedded interface. For example, when an embedded action button is not appropriate in the local application, you may want to make it invisible or disable it.

Online example: Subcanvas3Example

1. Before installing a subapplication using client:spec:, initialize the subapplication's builder to nil.

   This causes the subapplication to release its old builder.

2. Ask the subapplication for widget's wrapper by sending wrapperAt:, with the **ID** of the desired widget.

```
showMethods
    | selectedClass subcanvas spec |
    selectedClass := Smalltalk at: self classNames selection.
    spec := List2Example interfaceSpecFor: #listSpec.

    "Install the method names as the collection in the list
    application."
    self listApplication list: selectedClass selectors asSortedCollection.

    "Set the subbuilder to nil to discard the old builder. This is only
    necessary when the application uses the builder later to access
    widgets."

    listApplication builder: nil.

    "Get the subcanvas and install the list application."
    subcanvas := self widgetAt: #subcanvas.
    subcanvas client: listApplication spec: spec.
```

```
"Disable the embedded buttons (just to show that we can)."
(listApplication wrapperAt: #addButton) disable.
(listApplication wrapperAt: #deleteButton) disable.
```

Subcanvas Events

A Subcanvas triggers events when its view scrolls due to activity via its scroll bars or the Subcanvas builds and displays a new subcanvas.

#changing

When a subcanvas is sent a client:, client:spec:, or client:spec:builder: message to have it host a new canvas, prior to the work of building and displaying the new view, the subcanvas triggers the #changing event. This event is only triggered once a subcanvas has been built and displayed the first time. Thus the #changing event is not triggered upon the initial creation of a subcanvas.

#changed

After a subcanvas is sent a client:, client:spec:, or client:spec:builder: message to have it host a new canvas, and the new view is completed being built and displayed, the subcanvas triggers the #changed event. This event is only triggered once a subcanvas has been built and displayed the first time. Thus the #changed event is not triggered upon the initial creation of a subcanvas.

#scrollLeft

If while manipulating a horizontal scroll bar in a subcanvas, the subcanvas to the left, the subcanvas triggers the #scrollLeft event.

#scrollRight

If while manipulating a horizontal scroll bar in a subcanvas, the subcanvas scrolls to the right, the subcanvas triggers the #scrollRight event.

#scrollUp

If while manipulating a vertical scroll bar in a subcanvas, the subcanvas scrolls up, the subcanvas triggers the #scrollUp event.

#scrollDown

If while manipulating a vertical scroll bar in a subcanvas, the subcanvas scrolls down, the subcanvas triggers the #scrollDown event.

Tab Control

Online example: TabControlExample

The TabControl is an alternative to the Notebook widget, providing a tabbed folder look. To change the contents of the subpane below the row of tabs, click a tab, or select the tab using cursor keys and press **Enter**.

The Tab Control shows two scroller buttons, if there are more tabs than can be displayed in the available header space. Press the buttons to scroll the tabs.

Compatibility with Notebook

The Tab Control is largely, but not completely, protocol-compatible with the Notebook widget.

In many cases, you can change from a Notebook to a Tab Control simply by editing the application's windowSpec, replacing the #NoteBookSpec symbol to #TabControlSpec. This conversion method works, for example, with Notebook1Example, Notebook4Example, and Notebook5Example.

One limitation is that Tab Control does not support secondary tabs, as does Notebook. For this reason, the above conversion does not work for Notebook2Example or Notebook3Example.

For an example specifically using the Tab Control, load the TabControlExample parcel.

Adding a Tab Control

1. Select the Tab Control on the Palette and drop it on the canvas.
2. In its **Aspect** property, enter an aspect name. Also enter an **ID**, or keep the default. You will need the ID to identify the selected tab.
3. Define the accessor method for the aspect.

   The model should be a SelectionInList on either Strings, which make up textual tab labels, or Associations of a VisualComponent to a (optional) String. In TabControlExample, this is handled in two methods:

```
tabs
    tabs isNil
        ifTrue:
            [(tabs := SelectionInList with: self labelArray)
                selectionIndex: 1.
            tabs selectionIndexHolder
                onChangeSend: #tabsChanged
                to: self].
    ^tabs

labelArray
    "Private - The list of tab names (and/or icons). See also #specArray."

    ^Array
        with: 'Appearance'
        with: self class colorsImage -> 'Colors'
        with: self class printerImage -> ''
```

4. Paint subcanvases for each page (See the appearanceSpec, colorsSpec, and fontsSpec class methods in TabControlExample for examples.)

   So the canvas can be identified by an index, which will be retrieved from labelArray, TabControlExample defines a specArray method:

```
specArray
    "Private - The list of associated sub canvaes. See also #labelArray."

    ^#(#appearanceSpec #fontsSpec #colorsSpec)
```

5. Define a method that will respond by changing subcanvases when the tab selection has changed. This message is sent when the SelectionInList changes, as is defined in the tabs method.

```
tabsChanged
    "Every time, a tab is changed, a new sub canvas gets installed."

    | index |
    index := self tabs selectionIndex.
    (self widgetAt: #tabbing)
        client: self
        spec: (self specArray at: index)
```

The change is made by the client:spec: message sent to the Tab Control widget, which is retrieved from the builder.

For additional options and approaches, see Notebook above.

Defining Initial Labels

The above example defines the set of tab labels in code. You have the option of defining labels directly in the **Details** property page. This can be convenient, especially when there is no need to add pages dynamically.

To add a label, enter it into the Entry Field below the List and click **Add**. To change a label, select it, edit it in the Entry Field, and click **Change**. To delete, click Delete. To change the order of lables, select a label and drag it to its new position in the list.

When you click **Apply**, your labels are shown in the canvas, unlike when you provide labels in code.

You access the label by index as before, and so can change subcanvases in the same way. So, within the Application Model, for a Tab Control with aspect name tabs, send:

```
self tabs selectionIndex
```

returns the index of the currently selected tab.

Placing an Icon on a Tab

Each tab label can be a string, an icon, or an icon and a string. To use an icon, the SelectionInList must be an Association. The tabs method invokes labelArray to provide the labels, as follows:

```
labelArray
    "Private - The list of tab names (and/or icons). See also #specArray."

    ^Array
        with: 'Appearance'
        with: self class colorsImage -> 'Colors'
        with: self class printerImage -> ''
```

If the label is only a string, then it is provided simply as a String, as in the case with the first tab, ‘Appearance’.

If the label includes an icon, then the label is specified by an Association between the icon and a String. The second tab label is an Association between the graphic returned by self colorsImage and the String ‘Colors’.

The third tab is an icon without a String, so the String is empty.

Tab Control Events

A Tab Control triggers events when a tab selection is changing, when one of its tabs gains or looses focus, and when its tabs scroll.

#gettingFocus
When a tab control receives focus to one of its tabs, by being clicked on by the mouse, or tabbed into by the keyboard, when none of the tabs have had focus, the tab control triggers the #gettingFocus event.

#losingFocus
When a tab control loses focus from one of its tabs, by being clicked or tabbed away from, to anything except another of its tabs, the tab control triggers the #losingFocus event.

#tabbed
When a tab control's tabs have focus, and the and the Tab key is pressed in request to have the focus to move to the next widget in the tab order, the tab control triggers the #tabbed event.

#backTabbed
When a tab control's tabs have focus, and the Back-Tab (Shift-Tab) key is pressed in request to have the focus move to the previous widget in the tab, the tab control triggers the #backTabbed event.

#scrollLeft
If all of the tabs in a tab control are not visible, and the left arrow button is pressed to scroll additional tabs into view, then the tab control triggers the #scrollLeft event.

#scrollRight
If all of the tabs in a tab control are not visible, and the right arrow button is pressed to scroll additional tabs into view, then the tab control triggers the #scrollRight event.

#pageLeft
If all of the tabs in a tab control are not visible, and the left arrow button is pressed to scroll additional tabs into view while the Shift key is pressed, then the tab control triggers the #pageLeft event.

#pageRight

If all of the tabs in a tab control are not visible, and the right arrow button is pressed to scroll additional tabs into view while the Shift key is pressed, then the tab control triggers the #pageRight event.

#tabChanging

When a new tab is selected in a tab control, either by selection using the mouse or by keyboard navigation, before the view changes to the newly selected tab, the tab control triggers the #tabChanging event.

#tabChanged

After a new tab is selected in a tab control, either by selection using the mouse or by keyboard navigation, the tab control triggers the #tabChanged event.

Tables

A table displays data in a rows-and-columns structure. In appearance, tables are similar to dataset widgets, except that tables do not support direct editing, and tables can display dissimilar kinds of data. Data must be stored in a collection that allows two-dimensional access.

TableInterface

A table requires a relatively complex auxiliary object as its value model, which is provided by TableInterface. A TableInterface holds information about row and column labeling and formatting, in addition to the table data itself.

Within a TableInterface, the table data is held by a composite object, an instance of SelectionInTable, which stores the collection of cell contents and the selection index. The collection is expected to be a TwoDList (two-dimensional list), which converts a flat collection such as an array into a matrix of rows and columns. Alternatively, you can use a TableAdaptor to adapt a collection.

All of this interface machinery can be held by a single instance variable in the application model, and you can simply send messages to that object to fetch the table or the selection or any other aspect of it. However, you may find it economical to create instance variables to hold onto various aspects of the table interface. For example, the SelectionInTable is useful when your application model will need to access the contents of the table at run time.

Adding a Table

Online example: Table1Example

1. Add a table widget to the canvas.
2. On the **Basics** property page, in the **Aspect** property, enter a name for the aspect (e.g., tableInterface).

   Set other property checkboxes as desired.
3. Apply the properties and install the canvas, and use **define** to add the instance variable and accessor method for the aspect (tableInterface).

   Initialization is done in an initialize method (step 5), rather than by checking the **include initialization** option.
4. Use a code browser to add other instance variables, as appropriate.

   Because TableInterface is a complex object, it is useful to create additional instance variables for some of its components. In the

example we create one additional variable, sightingsTable and an accessor for it.

5 Create an initialize method to initialize the TableInterface.

A new TableInterface takes a SelectionInTable object, which can be initialized in the same method. In the example we set it as the value of sightingsTable.

Normally you wouldn't initialize the table with a hard-coded collection, but would gather the table data from a database or some other source.

```
initialize
    | list |
    super initialize.
    "Create a collection of sightings data."
    list := TwoDList
        on: #('Vulcans' 188 173 192 'Romulans' 26 26 452) copy
        columns: 4
        rows: 2.
    sightingsTable := SelectionInTable with: list.
    "Create a table interface and load it with the sightings."
    tableInterface := TableInterface new
        selectionInTable: sightingsTable.
```

Controlling Column Widths

All columns initially have an equal width that is determined by the space available in the table. If the table expands with the window, the column widths will also expand.

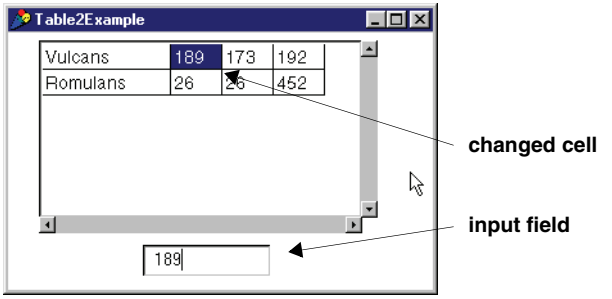
To set specific widths for the columns, send a columnWidths: message to the table interface. The argument is an array containing one number for each column. The number is the width in pixels. Any column for which no width is specified gets the width of the last entry in the array.

Reset the widths at any time by adding to the initialize method.

```
tableInterface columnWidths: #(100 40).
```

This expression changes the first column so that it is wide enough to show the entire name of the alien race. These widths will remain in effect even if the window is expanded.

Connecting a Table to an Input Field



A read-only table is sufficient for some applications, but in many situations the user needs a way to change the contents of a cell in the table. This can be arranged indirectly by placing an input field near the table and connecting it to the highlighted cell. This technique relies on a single cell being selected.

Online example: Table2Example

1. Add an input field to the canvas.
2. On the **Basics** property page. in the field's **Aspect** property, enter a name for its aspect (e.g., **cellContents**).
3. Apply the change and install the canvas, and use **define** to add an instance variable and accessor method for the aspect.
4. Add an instance method to update the table when the entry field changes.

   In the example we add changedCell in a change messages protocol:

```
changedCell
    | cellLocation |
    "Get the coordinates of the highlighted cell."
    cellLocation := self sightingsTable selectionIndex.
    "If a cell is selected, update its contents from the input field."
    cellLocation = Point zero
        ifFalse: [self sightingsTable table
            at: cellLocation
            put: self cellContents value]
```

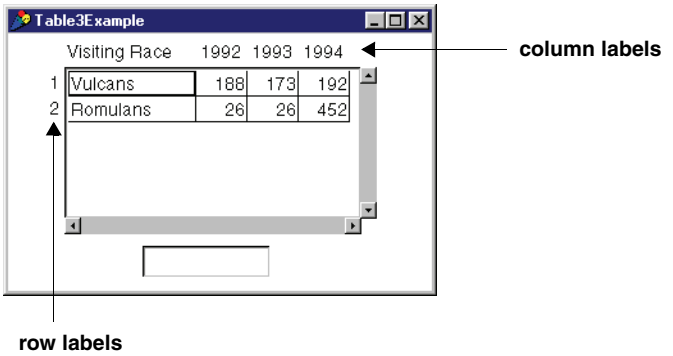
5. In the application model's initialize method, initialize the input field so it sends the changed message.

```
initialize
    | list |
    super initialize.
    "Create a collection of sightings data."
    list := TwoDList
        on: #('Vulcans' 188 173 192 'Romulans' 26 26 452) copy
        columns: 4
        rows: 2.
    sightingsTable := SelectionInTable with: list.
    "Create a table interface and load it with the sightings."
    tableInterface := TableInterface new
        selectionInTable: sightingsTable.

    cellContents := String new asValue.
    self cellContents onChangeSend: #changedCell to: self.
```

The application adds the input data into the selected cell.

Notice that when you select a new cell, its contents are not shown in the input field. To make the field update its contents when the table selection changes, you must register interest in the table selection with onChangeSend: and trigger an update in the input field.

Labeling Columns and Rows



You can label one or more columns by sending an array of labels to the table interface. For row labels, you need to send an array of labels and also an indication of the width of those labels.

Online example: Table3Example

This example adds code to the end of Table2Example's initialize method that initializes the row and column labels.

```
tableInterface
    columnLabelsArray: #('Visiting Race' '1992' '1993' '1994');
    rowLabelsArray: #(1 2);
    rowLabelsWidth: 20.
```

By default, all cells display their contents beginning at the left margin, and all labels are centered. You can align data and labels using any of three symbols: #left, #right, #centered, or #leftWrapped. Using these symbols, you can control the alignment of a column's data, a column's labels, or a row's labels.

Add code to the initialize method that initializes the label alignments.

```
tableInterface
    elementFormats: #(#left #right #right #right);
    columnLabelsFormats: #(#left #right #right #right);
    rowLabelsFormat: #right.
```

As the example shows, you can set row labels to the same alignment by passing a single symbol as argument, and the same applies to the column alignments. For column data and labels, however, you also have the option of setting each column's alignment individually, as we have done, by passing an array of symbols.

Table Events

A Table triggers events when a selection is changing, when it gains and looses focus, when it gets clicks from the mouse, and when it's view scrolls.

#clicked
When a table is clicked on with the <Select> mouse button, the table triggers the #clicked event.

#rightClicked
When a table is right clicked on with the <Operate> mouse button, the table triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the table.

#doubleClicked
When a table is double clicked on with the <Select> mouse button, the table triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus
When a table receives focus, either by being tabbed to or by clicking on by the mouse, the table triggers the #gettingFocus event.

#losingFocus

When a table loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the table trigger the #losingFocus event.

#tabbed

When a table has focus, and the and the Tab key is pressed to move the focus to the next widget in the tab order, the table triggers the #tabbed event.

#backTabbed

When a table has focus, and the Back-Tab (Shift-Tab) key is pressed to move the focus to the previous widget in the tab order, the table triggers the #backTabbed event.

#scrollLeft

If while navigating with the keyboard or mouse, or manipulating a horizontal scroll bar in a table, the table scrolls to the left, the table triggers the #scrollLeft event.

#scrollRight

If while navigating with the keyboard or mouse, or manipulating a horizontal scroll bar in a table, the table scrolls to the right, the table triggers the #scrollRight event.

#scrollUp

If while navigating with the keyboard or mouse, or manipulating a vertical scroll bar in a table, the table scrolls up, the table triggers the #scrollUp event.

#scrollDown

If while navigating with the keyboard or mouse, or manipulating a vertical scroll bar in a table, the table scrolls down, the table triggers the #scrollDown event.

#rowSelectionChanged

If a table has its selection mode set to Row, and while navigating with the mouse or keyboard, the selected row changes, the table triggers the #rowSelectionChanged event.

#columnSelectionChanged

If a table has its selection mode set to Column, and while navigating with the mouse or keyboard, the selected column changes, the table triggers the #columnSelectionChanged event.

#cellSelectionChanged

If a table has its selection mode set to Cell, and while navigating with the mouse or keyboard, the selected cell changes, the table triggers the #cellSelectionChanged event.

Text Editors

A text editor is useful for displaying and editing text that does not fit comfortably within a field, especially when the text is expected to have multiple lines. The text editor has built-in facilities for:

- Line wrapping
- Changing the text style
- Cutting, copying, and pasting
- Undoing and reverting
- Searching and replacing
- Printing
- Executing Smalltalk expressions

Adding a Text Editor

Online example: Editor1Example

1. Add a text editor to the canvas.
2. On the **Basics** property page, in the **Aspect** field, enter a name for the aspect.
3. Install the canvas, and use **define** to add an instance variable and aspect accessor method to the application model.
4. Create or edit the initialize method to set the aspect variable to a value holder containing the initial text to be displayed (usually an empty string).

```
initialize
    super initialize.

    comment := '' asValue.

    classes := SelectionInList with: Smalltalk classNames.
    classes selectionIndexHolder
        onChangeSend: #changedClass to: self.

    textStyle := #plain asValue.
    textStyle onChangeSend: #changedStyle to: self.

    readOnly := false asValue.
    readOnly onChangeSend: #changedReadOnly to: self
```

Retrieving and Modifying Selected Text

When the user highlights a portion of the text in an editor, your application can retrieve the highlighted text. Text editors frequently modify selected text in some way, such as to change the font, and then insert the revised text into the main text.

1. In an instance method, get the controller from the text editor widget, and ask the controller for the selected text.
2. Make any modifications to the text.
3. Ask the controller to replace the selection with a new text.
4. Ask the controller's view to reset its selections (to adjust for a possible width change in the selection).
5. Ask the view to redisplay itself.

```
changedStyle
    "A text style was selected -- apply it to the current selection in
    the comment."

    | c selectedText style |

    "Get the selected text."
    c := self controllerAt: #comment.
    selectedText := c selection.

    "If nothing is selected, take no action."
    selectedText isEmpty ifTrue: [^self].

    "If 'Plain' was selected, remove all emphases;
    otherwise add the new emphasis."
    style := self textStyle value.
    style == #plain
        ifTrue: [selectedText emphasizeAllWith: nil]
        ifFalse: [
            selectedText addEmphasis: (Array with: style)
                removeEmphasis: nil
                allowDuplicates: false].

    "Ask the controller to insert the modified text, then update the
    view."
    c replaceSelectionWith: selectedText.
    c view resetSelections.
    c view invalidate.
```

Highlighting Text

It is occasionally useful for the application to highlight a text selection, for example to indicate the result of a search.

To highlight text:

1. Write a method in the application model that gets the controller from the widget.
2. Ask the controller to select the text between two endpoints (and ask it to scroll the selection into view if necessary).
3. Ask the builder's component to take the keyboard focus, so the highlighting will be displayed.

```
changedClass
    "When the list selection changes, update the comment view."

    | selectedClass txt start wrapper |
    selectedClass := self classes selection.

    selectedClass isNil
        ifTrue: [self comment value: '' asText]
        ifFalse: [
            txt := (selectedClass asQualifiedReference value) comment.

            self comment
                value: txt.

            "Find and highlight the class name in the text."
            start := txt
                indexOfSubCollection: selectedClass asString
                startingAt: 1.
            start > 0 ifTrue: [
                wrapper := (self wrapperAt: #comment).
                wrapper widget controller
                    selectAndScrollFrom: start
                    to: start + selectedClass asString size - 1.
                wrapper takeKeyboardFocus]].
```

Aligning Text

By default, text in an editor is aligned at the left margin. For word-processing applications, you may want to center the text or align it at the right margin. You can change the alignment by setting the text editor's **Align** property.

For the initial alignment setting, use the **Details** property page and set the editor's **Align** property to **Left**, **Center**, or **Right**.

> **Note:** Alignment applies to the entire text. It cannot be applied selectively to a portion of the text.

To change the alignment for the text editor widget programmatically:

1. In a method in the application model, get the widget from the builder.
2. Get *a copy of* the widget's text style. (Do not modify the widget's text style directly, because that object is shared by many text editors in the system.)
3. Set the alignment of the text style to 0, 1, or 2 (0 is flush left, 1 is flush right, and 2 is centered).
4. Install the new text style in the widget.
5. Ask the widget to redisplay itself.

```
alignRight
    | widget style |
    widget := self widgetAt: #comment .
    style := widget textStyle copy.
    style alignment: 1.
    widget textStyle: style.
    widget invalidate.
```

Text Editor Events

A Text Editor triggers events when its value is changing, when it gains and looses focus, when it gets clicks from the mouse, and when it's view scrolls.

#changing
When the value of a text editor is about to be accepted after directly editing the value and exiting the input field, the text editor triggers the #changing event.

#changed
After the value of the text editor has been accepted, the text editor triggers the #changed event. To trigger #changed at each keystroke, send continuousAccept: to the widget's controller with the value true.

#clicked
When a text editor is clicked on with the <Select> mouse button, the text editor triggers the #clicked event.

#rightClicked

When a text editor is right clicked on with the <Operate> mouse button, the text editor triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the text editor.

#doubleClicked

When a text editor is double clicked on with the <Select> mouse button, the text editor will trigger the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus

When a text editor receives focus, either by being tabbed to or by clicking on by the mouse, the text editor triggers the #gettingFocus event.

#losingFocus

When a text editor looses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the text editor triggers the #losingFocus event.

#tabbed

When a text editor has focus, and the Control-Tab key is pressed to move the focus to the next widget in the tab order, the text editor triggers the #tabbed event.

#scrollLeft

If while editing, navigating with the keyboard, selecting text with the keyboard or mouse, or manipulating a horizontal scroll bar in a text editor, the view scrolls to the left, the text editor triggers the #scrollLeft event.

#scrollRight

If while editing, navigating with the keyboard, selecting text with the keyboard or mouse, or manipulating a horizontal scroll bar in a text editor, the view scrolls to the right, the text editor triggers the #scrollRight event.

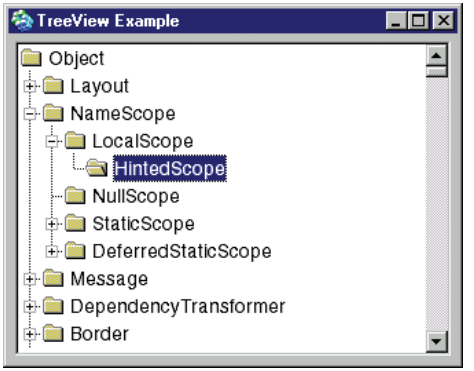
#scrollUp

If while editing, navigating with the keyboard, selecting text with the keyboard or mouse, or manipulating a vertical scroll bar in a text editor, the view scrolls up, the text editor triggers the #scrollUp event.

#scrollDown

If while editing, navigating with the keyboard, selecting text with the keyboard or mouse, or manipulating a vertical scroll bar in a text editor, the view scrolls down, the text editor triggers the #scrollDown event.

Tree View



Online Examples: TreeViewExample, AdvancedTreeViewExample

A Tree View provides an expandable list view on a hierarchy of items, such as a file directory system or an object hierarchy. The tree is expandable either by menu selection, mouse click, or key command, depending on its configuration.

A double-click with the mouse on an item expands/contracts that item. Keyboard control is described in the following table:

Key command	Description
Home	Select first item
PageUp	Move selection one page up
Up	Select previous item
Ctrl+Up	SelectionInTree: Select previous item at same level MultiSelectionInTree: Add previous item to selection list
Down	Select next item
Ctrl+Down	SelectionInTree: Select next item at same level MultiSelectionInTree: Add next item to selection list
PageDown	Move selection one page down
End	Select the last item
Left	Contract item
-	Contract item
Right	Expand item
Shift+Right	Expand item and sub items

Key command	Description
Ctrl+Right	Expand (invisible) root and all sub items
+	Expand item
*	Expand item and sub items
Ctrl+W	Reset buffer for string search
any	Do string search

Basics

The properties on the **Basics** page are the usual ones.

Aspect

The name of the aspect instance variable and accessors. The aspect must be either a SelectionInTree or a MultiSelectionInTree.

Details

The properties on the **Details** page are the same as for List, and are mostly self-explanatory.

Minimum Element Height

By default, the spacing of the lines in the tree list is determined by the font height. You can specify a minimum height as an Integer number of pixels. This may be necessary, for example, if you use an icon that is taller than the font, which would otherwise get cut off in display.

Whole Line Selection

By default, only the text is highlighted when an item in the Tree View is selected. To select the whole line, including the icon, if any, check the **Whole Line Selection** property.

Advanced

The **Item Text Emphasis** items specify method selectors to specify text formatting of a node label when the node is opened, closed, or a leaf node.

Items

The **Items** page is specific to Tree View.

Display String

This property sets the selector to be sent to an item in order to retrieve its textual representation.

Icons

None: Items will be displayed using the textual representation only.

Folder: Folder icons will be used.

User Defined: The selector that returns an item's iconic representation.

Visibility

Child Expansion: If checked, the TreeView will have a [+] image next to child items, if expandable. If unchecked (the default), it will not have a [+] next to the root item.

Root Expansion: If checked, the Tree View will have a [+] image next to the Root item. If unchecked (the default), it will not have a [+] next to the root item. Note that without the image, it is the application's responsibility to show the root expanded.

Lines: If checked, includes connecting lines between parent and child nodes.

In-Place Edited Selector: The selector that is used to notify an item about a successful in-place edit. The corresponding method must take one argument, which will be bound to the editing result (a String).

Notification

D. Click: By default, an element expands/contracts when double-clicked. You can provide a selector in this field to provide special handling in the case of a double-click. However, the default expansion/contraction will not occur and must be provided, if desired, by the method. To simply disable expansion/contractraction, enter #yourself in this field.

Initializing a Tree View

A TreeView's model is a SelectionInTree or a MultiSeletionInTree, which holds a TreeModel. The TreeModel is initialized during the interface building process.

There are basically two ways to initialize a TreeView's model. You can assign a proper model. For example:

```
initialize
    self treeView list: (TreeModel on: TreeTestNode example1)
```

Alternatively, you can send a root:displayIt:childrenBlock: message to an initialized TreeModel (see the example1 class method in TreeViewExample):

```
| app |
app := self new.
self openOn: app.
app treeView list
    root: Object
    displayIt: true
    childrenBlock: [:cls | cls subclasses select: [:ea | ea isMeta not] ]
```

For variations, refer to the initialize-release and instance creation method categories in TreeModel. Also, see the postBuildWith: method in AdvancedTreeViewExample for a more complex example.

Programmatic Interface

Controller Interface

The TreeController supports these local menu items: expand, expandFully, contractFully, and inplaceEdit.

The expand/contract methods operate on the currently selected item, the current selectionIndex.

Also, Tree Views do not automatically expand upon opening. To expand fully on open, define a postOpenWith: method in the application, such as this:

```
postOpenWith: aBuilder

    | treeWidget |
    treeWidget := self widgetAt: #TreeView1.
    treeWidget selectionIndex: 1.
    treeWidget controller expandFully.
```

This opens the view with the first item selected. If you do not want to leave an item selected, append this line:

```
treeWidget selectionIndex: 0
```

If the aspect is a MultiSelectionInTree, the expand/contract message is applied to the first element in its OrderedCollection of elements. The order of the elements is the order in which the elements were selected.

Model Interface

The widget's model is a TreeModel. TreeModel has an extensive public interface, including the following methods to specify the root of a tree and its children, and several expansion methods. Browse the class to see the full collection of methods.

Instance methods (sample only):

allChildrenFor: *anObject*
Answer all of the children of *anObject*, without regard to whether they are visible or not, or even whether *anObject* is visible or not. If the object is not somewhere in the tree, answer an empty collection, as if there are none.

allParentsFor: *anObject*
Answer all parents of *anObject*, returning the parent even if the child is not visible.

expand: *anIndex*
Expand the element at *anIndex* if it is not expanded yet. Also expand children that were expanded when this node was contracted.

expandFully: *anIndex*
Expand the element at *anIndex* and all of its children.

expandToLevel: *anInteger*
Expand all nodes to *anInteger* level.

root: *anObject*
Set the root node and display it

root: *anObject* **displayIt:** *aBoolean*
Set the root node and determine whether the root itself should be displayed or not

root: *anObject* **displayIt:** *aBoolean* **childrenBlock:** *aBlock*
Set the childrenBlock and the root node and determine whether the root itself should be displayed or not

Class methods:

on: *root*
Provide a new instance initialized with the given root.

on: *root* **displayRoot:** *displayIt*
Provide a new instance initialized with the given root. The second parameter determines whether the root will be displayed.

on: *root* **displayRoot:** *displayIt* **childrenWith:** *childBlock*
Provide a new instance initialized with the given root and childBlock. The second parameter determines whether the root will be displayed.

Adding a Tree View

1 Use a Palette to add a Tree View widget to the canvas. Leave the list selected.

2 On the **Basics** property page, fill in the list's **Aspect** property with the name of the method that will return an instance of SelectionInTree.

3 **Apply** the change, install the canvas, and use **define** to add an instance variable and aspect accessor method to the application model. This instance variable will hold the SelectionInTree.

4 Write an initialize method to initialize the SelectionInTree (see "Initializing a Tree View" above).

Adding Text Emphases

To add text formatting to tree node labels, do the following:

1 Select the TreeView widget on the canvas.

2 On the **Advanced** page in the GUI Painter Tool, enter a selector name for each of the node conditions that you want to specify an emphasis for. For example, to specify formatting for a closed node, enter a selector name in the **Opened Selector** field. The selectors may be either unary or single-argument keyword method names.

3 Install the canvas, and use **Define** to generate the new selectors.

4 Browse your application model's **tree view emphasis** method category, and edit the stub methods.

The methods must return an Array of text emphases. For example, if the unary selector is closedNode, and the emphasis is #italic, the method would be:

```
closedNode

    ^#(#italic)
```

Standard text emphases are #bold, #italic, #serif, #underline, #strikeout, #large, and #small.

To specify a color, include in the Array an association with #color as the key and a ColorValue as value:

```
closedNode

    ^Array with: #italic with: #color->ColorValue blue
```

If the selector is a single-argument keyword, the TreeView instance is passed into the method, and is available for testing. For example:

```
leafEmphasis: aTreeValue

    aTreeValue == VisualBlock ifTrue:
        [^Array with: #bold with: #color->ColorValue blue].
    ^#(#normal)
```

Tree View Events

A Tree View triggers events when a selection is changing, when the underlying list itself is changing, when it gains and loses focus, when it gets clicks from the mouse, when a node is expanded or collapsed, when an item in its list is edited in place, and when it's view scrolls.

#clicked
When a tree view is clicked on with the <Select> mouse button, the tree view triggers the #clicked event.

#rightClicked
When a tree view is right clicked on with the <Operate> mouse button, the tree view triggers the #rightClicked event. This event will occur without regard to whether there is a popup menu associated with the tree view.

#doubleClicked
When a tree view is double clicked on with the <Select> mouse button, the tree view triggers the #doubleClicked event. This event is always immediately preceded by a #clicked event.

#gettingFocus
When a tree view receives focus, either by being tabbed to or by clicking on by the mouse, the tree view triggers the #gettingFocus event.

#losingFocus
When a tree view loses focus, either by being tabbed away from, or by having another widget on the canvas gain focus, the tree view triggers the #losingFocus event.

#tabbed
When a tree view has focus, and the Tab key is pressed to move the focus to the next widget in the tab order, the tree view triggers the #tabbed event.

#backTabbed
When a tree view has focus, and the Back-Tab (Shift-Tab) key is pressed in request to have the focus move to the previous widget in the tab order, the tree view triggers the #backTabbed event.

#scrollLeft

If while navigating with the keyboard or manipulating a horizontal scroll bar in a tree view, the tree view scrolls to the left, the tree view triggers the #scrollLeft event.

#scrollRight

If while navigating with the keyboard or manipulating a horizontal scroll bar in a tree view, the tree view scrolls to the right, the tree view triggers the #scrollRight event.

#scrollUp

If while navigating with the keyboard or manipulating a vertical scroll bar in a tree view, the tree view scrolls up, the tree view triggers the #scrollUp event.

#scrollDown

If while navigating with the keyboard or manipulating a vertical scroll bar in a tree view, the tree view scrolls down, the tree view triggers the #scrollDown event.

#selectionChanging

When an item in a tree view is selected or unselected, or in the case of a multiple select tree view, when the selections change, before the change is applied, the tree view triggers the #selectionChanging event.

#selectionChanged

After an item in a tree view is selected or unselected, or in the case of a multiple select tree view, when the selections have changed, the tree view triggers the #selectionChanged event.

#selectionListChanged

Whenever the list underlying a tree view is manipulated, either by changing a value, reordering, adding or removing items, or changing the list as a whole, the tree view trigger the #selectionListChanged event.

#itemExpanded

When an item in the tree view is visually toggled from its collapsed view to its expanded view, the tree view triggers the #itemExpanded event.

#itemCollapsed

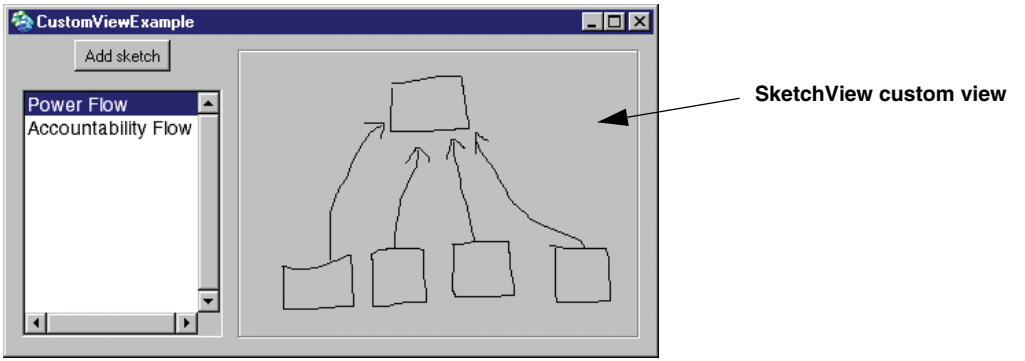
When an item in the tree view is visually toggled from its expanded view to its collapsed view, the tree view triggers the #itemCollapsed event.

#itemEdited

If an item in a tree view has an in place editor defined, and that item is edited, once the editing is completed the tree view triggers the #itemEdited event.

View Holder

The View Holder widget places a custom view on the canvas. While the interface to the View Holder widget is very simple, defining the custom view can be quite complex. See Chapter 8, “Custom Views”, for more information.



Adding a View Holder

Adding a view to a canvas simple:

1. Add a View Holder widget to the canvas.
2. In the View Holder’s **View:** field, enter the selector for method that returns an instance of the view, for example, **#myView**. Accept the change and install the canvas.
3. In your application model, define the method you entered in step 2. The method returns an instance of the custom view. For example:

```
myView

    ^MyViewClass new.
```

View Holder Events

A View Holder triggers events when its view scrolls due to activity via its scroll bars.

#scrollLeft
If while manipulating a horizontal scroll bar in a view holder, the tree view holder to the left, the view holder triggers the #scrollLeft event.

#scrollRight

If while manipulating a horizontal scroll bar in a view holder, the view holder scrolls to the right, the view holder triggers the #scrollRight event.

#scrollUp

If while manipulating a vertical scroll bar in a view holder, the view holder scrolls up, the view holder triggers the #scrollUp event.

#scrollDown

If while manipulating a vertical scroll bar in a view holder, the view holder scrolls down, the view holder triggers the #scrollDown event.

Index

D

E

Method Index