Developer Magic™: RapidApp™ User's Guide

Document Number 007-2590-001

CONTRIBUTORS

Production by Laura Cooper
Engineering contributions by Doug Young
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

Developer Magic™: RapidApp™ User's Guide
Document Number 007-2590-001

Contents

List of Examples

List of Figures

About This Guide

This book explains how to use the RapidApp™ application builder, a component of the Developer Magic™ Application Development Environment for developing applications to run on Silicon Graphics® workstations. This integrated development environment provides tools for rapid application development.

What This Guide Contains

This book contains the following chapters:

- Chapter 1, “Getting Started With RapidApp,” gives an overview of RapidApp, describes basic interaction techniques, and provides a simple example of creating an application using RapidApp.
- Chapter 2, “Creating Applications With RapidApp,” describes the process of developing an application using RapidApp, as well as giving details about RapidApp features such as code management and integration with other Developer Magic tools.
- Chapter 3, “Building Interfaces With RapidApp,” provides detailed information about choosing and using *interface elements* to build your application’s interface.
- Chapter 4, “Example Programs,” constructs some examples programs with RapidApp to demonstrate some of its features.
- Appendix A, “Frequently Asked Questions about RapidApp,” is a list of frequently asked questions (FAQs) and answers about RapidApp operation.
- Appendix B, “RapidApp Reference,” is a reference to RapidApp menus and palettes.

- Appendix C, “Source Code for the Calculator Application,” shows the source code for a simple example program developed throughout the book.
- Appendix D, “RapidApp Makefile Conventions,” documents the format of the *Makefile* the RapidApp generates.
- Appendix E, “VkEZ Reference,” documents the VkEZ convenience interface.

What You Should Know Before Reading This Guide

Because RapidApp covers many areas of application development and integrates with several Developer Magic tools and libraries, there are many topics with which you should be somewhat familiar to use RapidApp to its fullest capacity. For more information on these topics, consult the references provided in “Suggested Reading.”

This guide assumes that you are familiar with C++ and object-oriented programming. It also assumes that you have some knowledge of the IRIS IM™ toolkit, the Silicon Graphics port of the industry-standard OSF / Motif® interface toolkit.

Applications you develop should follow the Silicon Graphics guidelines for application interface design and should integrate into the Indigo Magic™ Desktop environment. In many places, RapidApp does this for you automatically. However, this guide assumes that you are familiar with these guidelines.

RapidApp links into other Developer Magic tools for building, analyzing, and debugging your application. This guide assumes that you know the basic purpose of these tools, but does not require in-depth knowledge of their use. The more you know about these tools, the quicker you can develop applications with RapidApp.

Some of the components that RapidApp allows you to incorporate in your application require knowledge of specific Silicon Graphics development libraries such as OpenGL™ and Open Inventor™. This guide assumes that you are already familiar with the underlying libraries if you decide to use these components.

Suggested Reading

RapidApp generates C++ code, and this guide assumes that your are familiar with C++ and object-oriented programming. The following manuals provide reference information about the Silicon Graphics implementation of the C++ language. These books are available online on the IRIS Insight™ SGI_Developer bookshelf:

- *C++ Language System Overview* contains an overview of newer language features of C++. Most of the extensions take the form of removing restrictions on what can be expressed in C++.
- *C++ Language System Product Reference Manual* contains a general description of the C++ language.
- *C++ Programming Guide* describes how to use the Silicon Graphics C++ compiler environment.
- *C++ Language System Library* discusses the iostream support in the C++ library and describes a data-type complex that provides the basic facilities for using complex arithmetic in C++.

The C++ classes generated by RapidApp are based on the IRIS ViewKit™ interface toolkit. This guide describes the features of IRIS ViewKit that you need to use the generated classes. If you want more information on IRIS ViewKit, you can consult the following book available online on the IRIS Insight SGI_Developer bookshelf:

- *IRIS ViewKit Programmer's Guide* provides detailed information about IRIS ViewKit class structure, features provided by the classes, and IRIS ViewKit programming techniques.

The following book describes the general approach used by the IRIS ViewKit library:

- Young, Doug. *Object-Oriented Programming with C++ and OSF/Motif.* Englewood Cliffs: Prentice Hall, Inc., 1992.

The actual user interfaces generated by RapidApp use the IRIS IM™ toolkit, the Silicon Graphics port of the industry-standard OSF / Motif interface toolkit. This guide assumes that you are familiar with the IRIS IM and Xt toolkits. For more information on IRIS IM, OSF / Motif, and Xt, you can consult the following books available online on the IRIS Insight SGI_Developer bookshelf:

- *OSF/Motif Programmer's Guide, Revision 1.2* is a guide to programming the various components of the OSF / Motif environment: the toolkit, window manager, and user interface language. Also available in printed form from Silicon Graphics and in bookstores: Open Software Foundation. *OSF/Motif Programmer's Guide, Revision 1.2*. Englewood Cliffs: Prentice-Hall, Inc., 1992.
- *OSF/Motif Programmer's Reference, Revision 1.2* documents the OSF / Motif commands and functions. Also available in printed form from Silicon Graphics and in bookstores: Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.2*. Englewood Cliffs: Prentice-Hall, Inc., 1992.
- *IRIS IM Programming Notes* describes the additional functionality provided by IRIS IM beyond that provided by OSF / Motif, as well as advice for Xt and Xlib programmers about programming in the Silicon Graphics X environment, including how to work with nondefault visuals.
- *The X Window System, Volume 4: X Toolkit Intrinsics Programming Manual* describes how to write X Window System™ programs using the Xt Intrinsics library. Also available in printed form from Silicon Graphics and in bookstores: Nye, Adrian and Tim O'Reilly. *The X Window System, Volume 4: X Toolkit Intrinsics Programming Manual, OSF/Motif 1.2 Edition for X11, Release 5*. Sebastopol: O'Reilly & Associates, Inc., 1992.

RapidApp provides significant support for following Silicon Graphics guidelines for application interface design and for automatically integrating your application with the Indigo Magic Desktop environment. For more information on following the Silicon Graphics interface style guidelines and integrating into the Indigo Magic Desktop environment, consult the following books available online on the IRIS Insight SGI_Developer bookshelf:

- *Indigo Magic User Interface Guidelines* contains recommended guidelines to help you design products that are consistent with other Silicon Graphics applications and that integrate seamlessly into the Indigo Magic Desktop environment.
- *Indigo Magic Desktop Integration Guide* is a companion to the *Indigo Magic User Interface Guidelines* that explains how to integrate applications into the Indigo Magic Desktop environment.

- *Software Packager User's Guide* describes how use Software Packager, a graphical tool for packaging software for installation on Silicon Graphics workstations. Products packaged with Software Packager can be installed with Software Manager, an Indigo Magic Desktop utility for installing software.

RapidApp links into other Developer Magic tools for building, analyzing, and debugging your application. For more information on these tools, consult the following book, available online on the IRIS Insight SGI_Developer bookshelf:

- *Developer Magic: ProDev WorkShop and MegaDev Overview* gives you broad exposure to the ProDev WorkShop tools as well as pointers to the documentation for getting detailed information.

The following books, available online on the IRIS Insight SGI_Developer bookshelf, describe specific Silicon Graphics development libraries underlying some specific components that you can incorporate in your application:

- *The Inventor Mentor* introduces graphics programmers and application developers to Open Inventor, an object-oriented 3D toolkit. Also available in printed form from Silicon Graphics and in bookstores: Wernecke, Josie. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2.* Addison-Wesley Publishing Company, 1992.
- *OpenGL Programming Guide* describes how to use OpenGL, allows you to create interactive programs that produce color images of moving three-dimensional objects. Also available in printed form from Silicon Graphics and in bookstores: Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison-Wesley Publishing Company, 1994.

Also there are several books available commercially that you might find useful in learning IRIS IM (OSF/Motif) and Xt programming techniques, including:

- Young, Doug. *The X Window System, Programming and Applications with Xt, OSF/Motif Edition,* Second Edition. Englewood Cliffs: Prentice Hall, Inc., 1994.

- George, Alistair. *Advanced Motif Programming*. Englewood Cliffs: Prentice Hall, Inc., 1994.

Font Conventions in This Guide

These style conventions are used in this guide:

- **Boldfaced text** indicates that a term is an option flag, a data type, a keyword, a function, or an X resource.
- *Italics* indicates that a term is a filename, a button name, a variable, an IRIX command, a document title, or an image or subsystem name.
- "Quoted text" indicates menu items.
- `Screen type` is used for code examples and screen displays.
- **`Bold screen type`** is used for user input and nonprinting keyboard keys.
- Regular text is used for menu and window names, and for X properties.

Upgrading to Builder Xcessory

RapidApp is adapted from Integrated Computer Solution's powerful graphical user interface builder for OSF / Motif, Builder Xcessory™. Builder Xcessory offers all of the functionality provided by RapidApp, plus additional features including:

- Generation of C, UIL and Ada code in addition to C++.
- Full access to the entire set of Motif resources
- Support for adding new or custom widgets
- Several specialized editors including a hierarchical widget tree browser, a color editor, an integrated pixmap editor, and a fontlist editor.

For more information about the features and functionality of Builder Xcessory, call Integrated Computer Solutions (ICS) at (617) 621-0060 ext. 164, send email to *info@ics.com*, or visit the World Wide Web site *http://www.ics.com*.

Chapter 1

Getting Started with RapidApp

This chapter provides an overview to RapidApp and shows you how to start developing simple applications.

Chapter 1

Getting Started With RapidApp

This chapter provides an introduction to developing application with RapidApp. It contains:

- “RapidApp Overview,” an overview of RapidApp
- “Installing and Starting RapidApp,” instructions for installing and running RapidApp
- “The RapidApp Interface,” an overview of the RapidApp interface
- “Basic Interaction Techniques,” the basic techniques for building applications with RapidApp
- “Example: A Calculator,” a simple example
- “EZ Convenience Functions,” a discussion of a set of convenience functions for accessing widget values

RapidApp Overview

RapidApp is an interactive tool for creating applications. It integrates with other Developer Magic tools, including cvd, cvstatic, cvbuild, Delta C++, Smart Build, and others, to provide an environment for developing object-oriented applications as quickly as possible. RapidApp generates C++ code, with interface classes based on the IRIS ViewKit toolkit. Its predefined interface components facilitate your use of other Developer Magic libraries such as OpenGL and Open Inventor.

You can use RapidApp for constructing typical desktop applications in which the user interface has a significant effect on the overall application architecture. The applications produced by RapidApp are automatically integrated into the Indigo Magic Desktop environment, making RapidApp the easiest way to take advantage of most of Silicon Graphics’ interface and desktop technology.

When using RapidApp, you work with a combination of IRIS IM *widgets* and *components* based on IRIS ViewKit classes. This guide refers to widgets and components collectively as *interface elements*. You create, select, position, and manipulate interface elements using techniques similar to those supported by drawing editors such as IRIS Showcase™. You can move interface elements after creating them, and you can edit various attributes (known as *resources*) to change their appearance or behavior.

RapidApp provides a great deal of support for creating interactive applications, but it doesn't completely replace programmer expertise. Think of RapidApp as a sophisticated editor with domain-specific support for helping you create graphical user interfaces. Although RapidApp can greatly facilitate the task, you remain in control and must understand the tasks being performed.

To use RapidApp effectively, you should have a basic knowledge of IRIS IM, C++, IRIS ViewKit, and recommended Indigo Magic user interface guidelines. You don't have to be an IRIS IM expert, but a basic understanding of widget hierarchies, the behavior of IRIS IM manager widgets, resources, and callbacks is very helpful. Because RapidApp produces IRIS ViewKit programs, you should also understand the basic idea of user interface *components*, as well as be familiar with C++ classes and object-oriented concepts such as inheritance, polymorphism (virtual functions), and so on. Finally, knowledge of the Indigo Magic user interface guidelines helps you understand the type of application RapidApp helps you create. You can find references for all of these topics in "Suggested Reading" on page xxi.

Installing and Starting RapidApp

The *RapidApp Release Notes* contains complete instructions for installing RapidApp. To install and run RapidApp, your system must have the IRIS Development Option (IDO), which includes the C compiler and the X and IRIS IM development systems, and the C++ Development Option, which includes the IRIS ViewKit development system. To use the other ProDev WorkShop tools, such as cvd, cvstatic, and cvbuild, you must install the ProDev WorkShop products. To use special interface components that take advantage of other Developer Magic libraries such as Open Inventor, you must also install those development options. Consult the *RapidApp Release*

Notes for a complete list of products you must install on your system to install and run RapidApp.

To start RapidApp from a shell window, enter:

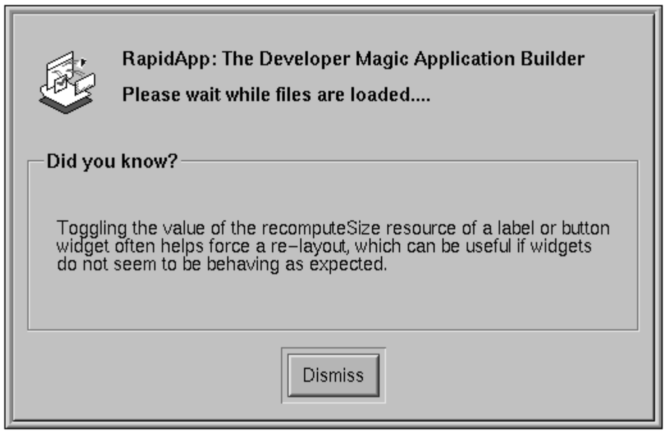
```
% rapidapp
```

Alternatively, you can go to the ToolChest and, in the Find menu, select “An Icon.” In the Find an Icon dialog, search for “rapidapp.” You can then drag the RapidApp icon to your desktop and launch the program by double-clicking on the icon.

The RapidApp Interface

RapidApp displays a startup screen when you invoke it. By default, the startup screen contains a random “tip,” a suggestion for how to use RapidApp better. Figure 1-1 shows the startup screen with an example tip.

You can dismiss this screen once the RapidApp main window appears. You can also set the RapidApp preferences so that RapidApp doesn’t display the tips or dismisses the startup screen automatically when the main window appears. To change the behavior of the startup screen, select “RapidApp Preferences” from the RapidApp Options menu. In the RapidApp Preferences dialog that appears, set the startup options that you want.



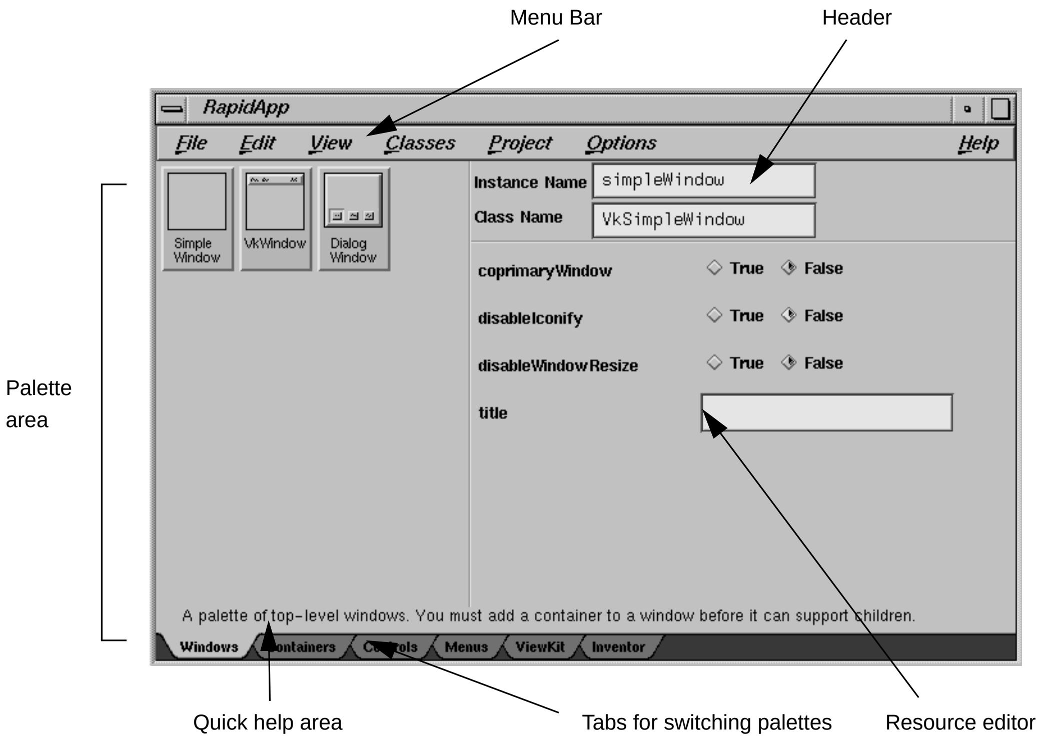
RapidApp: The Developer Magic Application Builder
Please wait while files are loaded....

Did you know?

Toggling the value of the recomputeSize resource of a label or button widget often helps force a re–layout, which can be useful if widgets do not seem to be behaving as expected.

Figure 1-1 RapidApp Startup Screen

Figure 1-2 shows the RapidApp main window. This window contains five main areas: the menu bar, the palette, the instance header area, the resource editor, and the quick help area. The following sections describe each of these areas.



Instance Name simpleWindow
Class Name VkSimpleWindow
coprimaryWindow ◇ True ◆ False
disableIconify ◇ True ◆ False
disableWindowResize ◇ True ◆ False
title

A palette of top-level windows. You must add a container to a window before it can support children.

Figure 1-2 RapidApp Main Window

The RapidApp Menu Bar

The RapidApp menu bar provides the following menus:

File	Allows you to save and open builder files. You can also quit RapidApp through the File menu.
Edit	Supports cut, copy, and paste operations, as well as some commands for manipulating a selected interface element.

View	Contains entries for switching between the default “Build Mode,” in which an interface can be constructed, and “Play Mode,” in which an interface can be tested.
Classes	Allows you to create and edit user-defined components.
Project	Contains entries that relate to the complete life cycle of a project’s development. The commands on this pane allow you to generate code, browse and edit files, build an application, run the program under a debugger, and so on.
Options	Allows you to set several options primarily related to how code is generated. The entries on this menu pane allow you to configure the file naming conventions, directory paths, and so on.
Help	Allows you to access the online help system.

The RapidApp Palette Area

The left side of RapidApp’s main window contains palettes of widgets and components that you can use to construct an interface. The area contains multiple palettes you can access via tabs that appear along the lower left side of the window. Figure 1-2 shows the Windows palette. The items on the Windows serve as top-level windows for your application. Clicking on the tab labeled “Controls” shows another palette that contains buttons, sliders, text entry areas, and other basic control widgets.

The RapidApp Instance Header

The instance header displays the instance name and the class name of the currently selected interface element. RapidApp automatically generates an instance name for an element when you create it. You can change the name of the element by entering a new string in the Instance Name field. RapidApp uses this name when it generates code for the element.

The class name is the widget class for IRIS IM *widgets* or the C++ class name for *components*. For some IRIS IM widgets, you can change the widget class and thus change the type of widget. For example, you can change a Label into a Push Button by changing the class name of the widget from XmLabel to XmPushButton.

The RapidApp Resource Editor

The resource editor occupies the right side of the RapidApp main window. The resource editor is initially empty when RapidApp first appears. When you create or select an interface element, this area displays lists of customizable parameters, known as *resources*, for the selected interface element. The contents of this area change dynamically, depending on the interface element selected.

The RapidApp Quick Help Area

The quick help area is immediately above the palette tabs. When you point with the mouse to an item in the RapidApp interface, the quick help area displays a one-line help message for the item.

Basic Interaction Techniques

The following sections show the basic interaction techniques for creating a user interface. The techniques you use to interact with RapidApp are similar to those you would use with a drawing tool such as Showcase. Of course, the RapidApp objects are more complex than lines and rectangles in simpler drawing tools and frequently have characteristics that affect how you interact with them. For example, an object you create in a drawing tool typically doesn't change its size or position, unless you explicitly change it, but many manager widgets that you create with RapidApp *can* change the size or position of elements they contain.

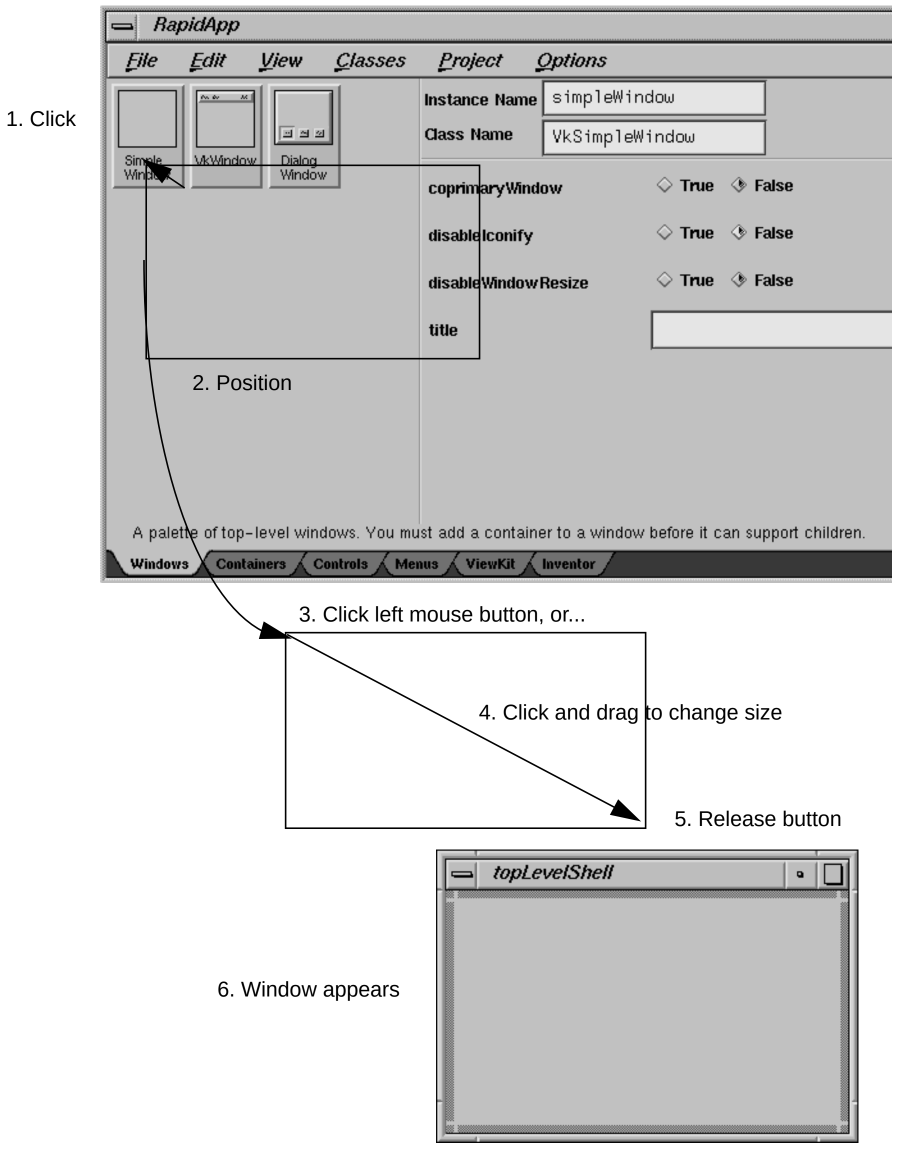
Creating Interface Elements

To create a new interface element, simply click the appropriate icon in the palette. A rubber-band box appears, representing the initial default size of the new widget. Move the cursor over the desktop and press the left mouse button to position the upper left corner of the widget. Then you can release the mouse button to accept the default size of the widget, or you can drag out a new size before releasing the mouse button.

Note: RapidApp enforces a minimum size of 20x20 pixels for all interface elements. Additionally, the window manager enforces minimum sizes for its direct children. So if you try to create a 20x20 button using the technique just described, the actual widget might be larger. In practice this is not a problem, because real interfaces seldom consist of a single small widget as a direct child of a shell. Furthermore, this behavior matches the behavior you would get from a running program.

Figure 1-3 demonstrates how to create a Simple Window:

1. Select the Simple Window icon from the Windows palette.
2. Position the rubber-band rectangle.
3. Press the left mouse button. Or, if you want to resize the Simple Window, drag rubber-band outline to the desired size before releasing the mouse button.
4. RapidApp creates the Simple Window when you release the mouse button.



Instance Name: simpleWindow
Class Name: VkSimpleWindow

coprimaryWindow: True [✓ False]
disableIconify: True [✓ False]
disableWindowResize: True [✓ False]
title:

A palette of top-level windows. You must add a container to a window before it can support children.

Figure 1-3 Creating a Widget

Adding a Container to a Top-Level Window

A top-level window such as a Simple Window can contain only one child element. Therefore, once you create a window, you must typically create a container within the window. For example, you could create a Bulletin Board container within the window as follows:

1. Click the Containers tabs to display the Containers palette.
2. Select the Bulletin Board icon.
3. Position the rubber-band rectangle so that the upper-left corner of the rectangle is within the Simple Window.
4. Press the left mouse button.

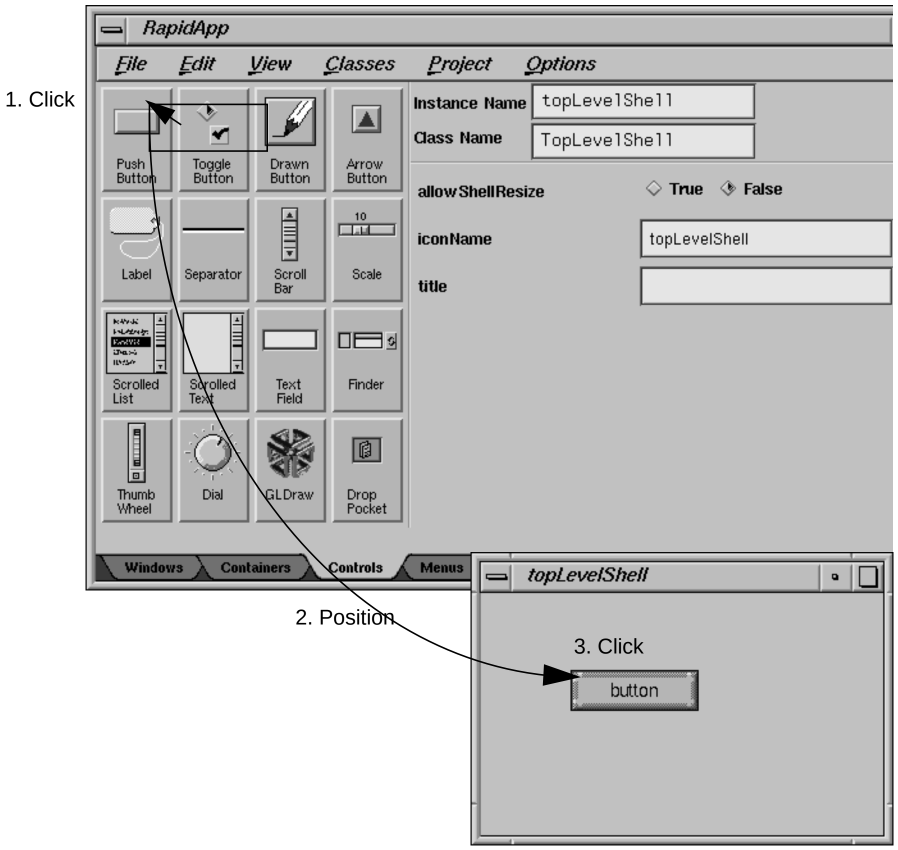
   RapidApp creates the Bulletin Board within the Simple Window when you release the mouse button. The Bulletin Board automatically resizes to fill the entire window.

Creating Interface Elements in an Existing Container

Once you have one or more *containers,* you can add other interface elements to them as *child elements.* The process is the same as creating an initial interface element, except that you position the new element to lie within the bounds of the parent. For example, Figure 1-4 shows how to create a Push Button widget within a Bulletin Board container:

1. Switch to the Controls palette.
2. Click the Push Button icon.
3. When the rubber-band rectangle appears, move the mouse so that the rubber-band box is positioned over the Bulletin Board at approximately the location you want to place the widget.
4. Press the left mouse button, and either release immediately or drag to change the button's initial size before releasing.

   RapidApp creates the button as a child of the Bulletin Board when you release the mouse button.



Instance Name: topLevelShell
Class Name: TopLevelShell
allowShellResize: True [✓ False]
iconName: topLevelShell
title:

Figure 1-4 Creating a Pushbutton as a Child of a Bulletin Board

You can also add children to a container by dragging an icon from the palette directly using the middle mouse button:

1. Press the middle mouse button over the item in the palette.
2. Drag the item to the desired parent container and release the mouse button.

   RapidApp creates the interface element as a child of the container when you release the mouse button.

Explicit Focus Mode

By default, RapidApp uses a pointer focus model when creating widgets: when you create a widget and drop it over a valid parent, the new widget becomes a child of that parent even if you had another parent widget selected.

Sometimes you might find it convenient to use explicit focus mode when adding child elements to a container. In this mode,RapidApp adds all child elements you create to the currently selected container, no matter where you drop the children on the screen. To turn on explicit focus mode, toggle on “Keep Parent” in the View menu.

As a further convenience, when explicit focus mode is on, RapidApp grays out all icons that you can’t add to the currently selected container. For example, if you select a menu bar when “Keep Parent” is toggled on, RapidApp grays out all elements other than the various menu panes that you can add to the menu bar.

Moving and Resizing Interface Elements

Once you’ve created an interface element, you might need to change its position or size. In RapidApp you can reposition or resize interface elements in three ways:

- For some IRIS IM container widgets, such as the Form and Bulletin Board widgets, the order in which you create the child elements is unimportant. You can place child elements anywhere within the container widgets, then reposition or resize them directly.
- For other IRIS IM container widgets, such as the RowColumn, Paned Window, and Spring Box widgets, the creation order of their children is significant. Furthermore, these containers control the geometry of their children more “tightly”; often you can’t resize child widgets individually. RapidApp provides support for repositioning child elements of this type of container more easily, but be aware that you’ll often have to manipulate the container widget itself (for example, by changing its resources) to affect the size or layout of its children.

- The last case of repositioning in RapidApp is to move a child element to another parent. RapidApp provides both a clipboard and a drag-and-drop mechanism for easily reparenting elements in your interface.

In all cases, the container determines the exact position (and often the size) of a child element. The builder allows you to manipulate an element interactively, but the actions the builder allows you to perform ultimately become requests to the element's container. This is a core part of the architecture of Xt and IRIS IM, and the builder cannot change this. If you have trouble, make sure you understand the layout algorithm supported by the container widget you are working with.

Directly Repositioning and Resizing Child Elements

For container widgets where widget creation order is not important, you can reposition a child element simply by dragging it using the left mouse button (see Figure 1-5). The child element "snaps" to positions along an invisible grid. You can control the resolution of the *snap grid* through the "Snap to Grid" option of the View menu. You can set the resolution to 2, 5, 10, or 20 pixels, or turn off the snap grid.

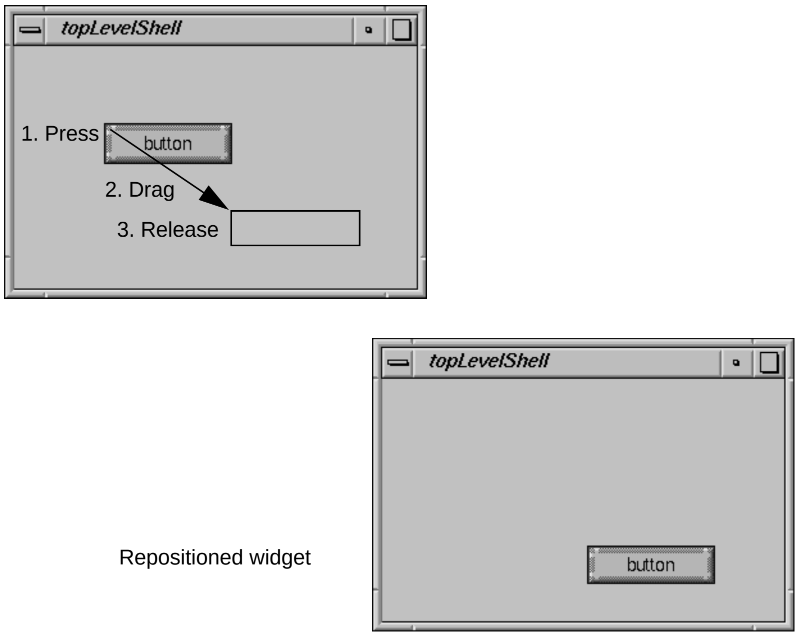


Figure 1-5 Repositioning a Widget in a Bulletin Board Container

Some containers have more complex behaviors. In a simple Bulletin Board widget, moving a child is equivalent to changing its *x,y* position, as determined by its **XmNx** and **XmNy** resources. However, in a Form widget, moving a component is equivalent to changing its **XmNleftOffset**, **XmNrightOffset**, **XmNbottomOffset**, and / or **XmNtopOffset** resources.

In addition to using the mouse, you can also use the arrow keys for fine positioning. Each time you press an arrow key, the child elements moves one pixel in the corresponding direction. The arrow keys ignore the snap grid setting; you can use them for fine-grained positioning regardless of the grid resolution.

To change an element's size in a container that allows free movement, simply select one of the handles that surround a selected element and drag a side or a corner until the element is the desired size. Figure 1-6 illustrates this process.

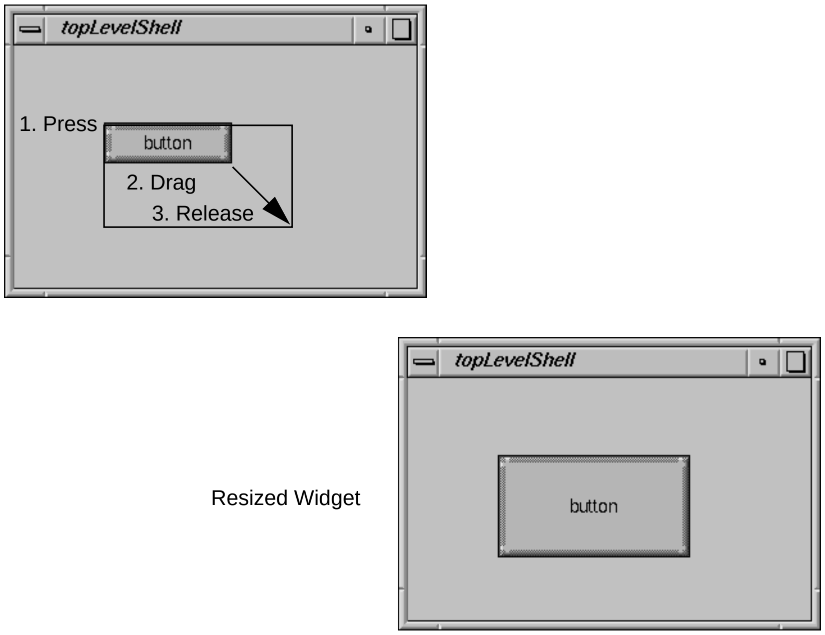


Figure 1-6 Resizing a Widget

If the element is too small to resize easily, you can select it, then select “Grow Widget” from the Edit menu (or use the **`<Ctrl+g>`** accelerator), which increases the width and height of the selected element by 20 pixels.

Indirectly Positioning Child Elements

For containers in which a child’s position depends on the creation order of the widget’s children, changing an element’s *x,y* position is meaningless. In many cases, though not all, it is also meaningless to try to resize a child of such a container. Although RapidApp could help you try to move the child, the container ignores the movement. For these containers, you can reposition each child within the container using the arrow keys or the “Up / Left” and “Down / Right” options in the Edit menu (or the **`<Ctrl+u>`** and **`<Ctrl+d>`** accelerators, respectively). The commands effectively alter the creation order of the element being moved and its siblings. Figure 1-7 shows an example of repositioning a toggle button in a RowColumn container.

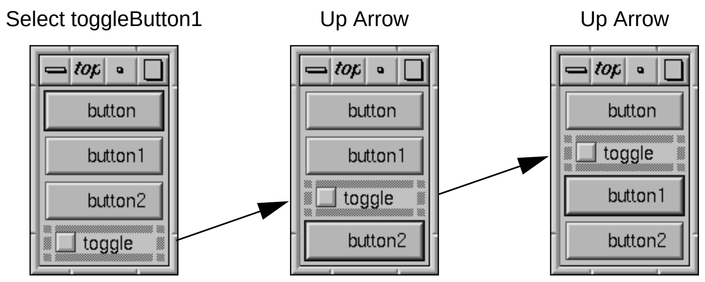


Figure 1-7 Repositioning a Widget in a RowColumn

Reparenting Child Elements

In addition to moving an element within the container in which you originally placed it, you might find it useful to move an element from one container to another. IRIS IM normally doesn't support *reparenting* widgets. However, in RapidApp, this operation is possible and can be done in two ways:

- You can cut a widget or widget hierarchy to the clipboard using "Cut" from the Edit menu (or the **`<Ctrl+x>`** accelerator), then paste it using "Paste" from the Edit menu (or the **`<Ctrl+v>`** accelerator). You can also copy an element or element hierarchy to the clipboard using "Copy" from the Edit menu (or the **`<Ctrl+c>`** accelerator).
- Alternatively, you can drag an element using the middle mouse button. This uses the IRIS IM drag-and-drop mechanism to effectively cut and paste between containers. If you hold down the shift key while dragging with the middle mouse button, RapidApp copies the selected elements instead of moving them.

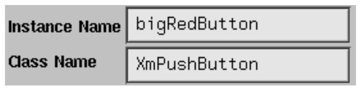
Deleting Interface Elements

You can delete an element by selecting it then selecting "Cut" from the Edit menu (or the **`<Ctrl+x>`** accelerator). Alternatively, you can select the element, then select "Delete" from the Edit menu (or the **`<Backspace>`** or **`<Delete>`** accelerators); however, doing so does not save the widget on the clipboard, so you can't paste it back afterwards. There is no undo feature.

Naming Interface Elements

When you initially create an interface element, RapidApp assigns a unique generated name. All interface element names in an application must be unique. For example, if you create a Push Button widget, RapidApp names it “button.” RapidApp names the next Push Button you create “button1,” and so on. The name determines both the string given to the widget (its resource name) when it is created, and the variable that represents the widget in the program.

You can change the name of an interface element at any time simply by editing the Instance Name field in the header area when the interface element is selected. Figure 1-8 shows the header area with a user-specified name displayed in the Instance Name field.



Instance Name	bigRedButton
Class Name	XmPushButton

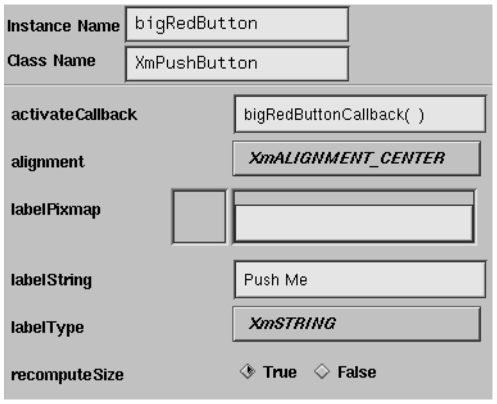
Figure 1-8 Header Area

Editing Interface Element Resources

In addition to changing the position and size of an interface element, you can control its appearance and behavior by setting *resources* supported by the element. For example, one resource of a Label widget determines the string it displays, and one resource of a RowColumn container widget determines how many rows or columns it creates.

IRIS IM widgets are highly configurable and typically include a large number of resources—even a simple label widget supports nearly 50 customizable resources—but typically you need to access only a few when writing an application. Therefore, RapidApp displays only the most commonly used resources. You can still access all resources programmatically by editing the source code generated by RapidApp or by editing the application’s resource file.

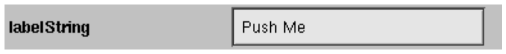
To see the resources available for a particular element, simply click the element with the left mouse button. The RapidApp resource editor area then displays the resources for that element. The names of the resources are listed along the left side of the resource editor area with the current value of each resource to the right. Figure 1-9 shows the resource editor area.



Instance Name: bigRedButton
Class Name: XmPushButton
activateCallback: bigRedButtonCallback( )
alignment: XmALIGNMENT_CENTER
labelPixmap
labelString: Push Me
labelType: XmSTRING
recomputeSize: ◆ True ◇ False

Figure 1-9 Resource Editor Area

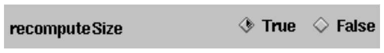
Resources can be of several different types: some are strings, some are Boolean values, and others are enumerated. The way RapidApp displays the value of a resource depends on its type. Figure 1-10 shows how RapidApp displays a resource with a string value. To change the value of this resource, simply edit the contents of the text field. When you modify the value, the text field changes color slightly. RapidApp accepts the new value when you press the **`<Return>`** key or click the mouse outside of the text field; RapidApp then changes the text field to its original color to indicate that it has accepted the value. If you enter an illegal value, RapidApp displays an error dialog and reverts the text field to its former value.



labelString: Push Me

Figure 1-10 A Resource Whose Value Is a String

Figure 1-11 shows how RapidApp displays a resource with Boolean values. To change a Boolean resource's value, simply click the desired toggle.



recomputeSize: ◆ True ◇ False

Figure 1-11 A Resource Whose Value Is Boolean

Figure 1-12 shows an enumerated resource. RapidApp displays the current value of this resource in the option menu to the right of the resource name. To change these resources, press and hold the left mouse button over the option menu to display a list of possible resource values, drag to select the desired value, and release the mouse button. RapidApp then displays the new value in the option menu.

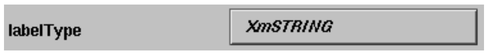


Figure 1-12 A Resource Whose Value Is Enumerated

Most resources are easy to use, but a few require an understanding of the selected element. The following discussion provides more information about various types of resources.

Callbacks

Callbacks are functions that associate program behavior with user input. For example, the PushButton widget has an **activateCallback** function that is called when the user clicks the button. As shown in Figure 1-13, to specify a callback function in RapidApp, simply type the name of a function in the text field beside the name of the callback. (You don't have to enter the parentheses; RapidApp automatically provides them when you finish editing the callback resource.) When RapidApp generates code, it creates these callback functions as empty virtual member functions in a C++ class. The implementation of the function body is left up to you. (See "Code Management" on page 40 for more information on editing generated code to implement functionality.)

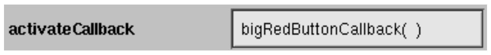
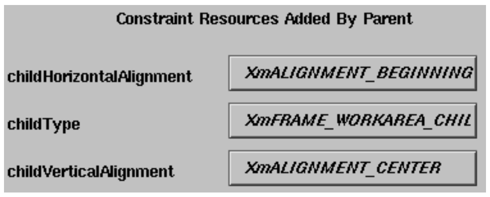


Figure 1-13 Adding a Callback

Constraints

IRIS IM supports the concept of *constraints*, which are resources that are added to an element when it is contained by a particular type of container. For example, when an element is the child of a Frame widget, the Frame adds the **childType** resource to the child, which determines the position of the child within the Frame. Therefore, you might see a constraint resource in one element and not in another of the same type if the elements are contained within different types of containers. The resource editor area lists constraint resources separately from other resources, below a label identifying them as constraint resources. You can modify constraint resources just as you do other resources. Figure 1-14 shows the constraint resources added to an XmLabel widget when contained by an XmFrame widget.


Constraint Resources Added By Parent

childHorizontalAlignment	XmALIGNMENT_BEGINNING
childType	XmFRAME_WORKAREA_CHIL
childVerticalAlignment	XmALIGNMENT_CENTER

Figure 1-14 Resource Editor Area, Showing Constraint Resources

Dynamic Resources

RapidApp also supports several dynamic resources that act much like constraints but that are not supported by IRIS IM. These correspond to extensions and features provided by IRIS ViewKit classes. For example, when you place a PushButton widget in a menu pane, RapidApp displays an **undoCallback** resource for the PushButton. This callback isn’t a resource supported by IRIS IM, but it provides support for the IRIS ViewKit undo mechanism.

RapidApp also determines when to display other resources. For example, the IRIS IM PushButton widget supports an accelerator resource that describes a key combination that users can type to activate the button when the button is in a menu. Although the PushButton widget supports the resource at all times, the resource is meaningless when the button isn’t in a menu. To ensure the proper use of this resource, RapidApp displays it only when the PushButton is in a menu pane.

Additional Interaction Techniques

This section describes some additional techniques for working with interface elements.

Locking on to an Element

Sometimes it's hard to manipulate elements because they're too close to or covered by other elements. For example, some container wrap "tightly" around their child elements. In these cases, it can be difficult to move or resize the container without accidentally selecting another element.

To "lock on" to an element and prevent RapidApp from selecting another element, simply hold down the **`<Ctrl>`** while manipulating the widget you have selected.

Selecting a Parent Element

Some elements can be hard to select because they are completely covered by one or more children. For example, you can't click on a shell widget, or a Simple Window that has a child. To access elements like these, you can select a child element of the desired container, then select "Select Parent" from the Edit menu (or use the **`<Ctrl+p>`** or **`<Ctrl+Shift+left mouse button>`** accelerators).

Viewing the Widget Hierarchy

For complex layouts, it's often useful to see the structure of the widget hierarchy you're creating. RapidApp doesn't have a built in widget tree view, but you can still examine the widget hierarchy using the *editres* program. Simply start *editres* and click on the window for which you want see the widget hierarchy.

Resetting IRIS IM Widgets

The IRIS IM widget set was developed long before interface builders began to appear, and the idea that widgets might be interactively created and manipulated wasn't considered in the design and implementation of IRIS IM. Not surprisingly, this makes working with IRIS IM in a builder somewhat more difficult than it might be if IRIS IM were designed to support such tools. In particular, IRIS IM has no way to completely reset a widget's state, so after a large number of changes, it can be hard to see what would happen if a widget was created initially in that state.

The closest most widgets can come to being reset is to be resized. Resizing a widget typically recomputes its layout, if it's a container, or its appearance, if it's a primitive widget. If a widget doesn't appear to be behaving as you expect, try a slight resize and watch what happens.

For example, consider the RowColumn widget in Figure 1-15.

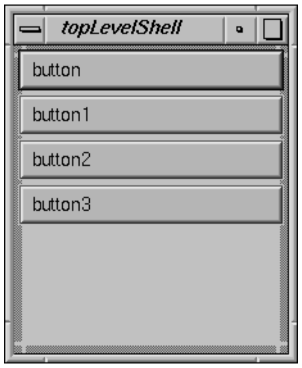


Figure 1-15 Initial RowColumn Layout

Now change the RowColumn resources so that **orientation** is XmHORIZONTAL, **packing** is XmPACK_COLUMN, and **numColumns is** 2. This produces the layout in Figure 1-16, which is incorrect because it isn't the layout that the RowColumn produces when you run your application.

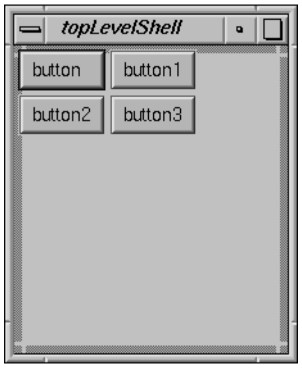


Figure 1-16 Incorrect RowColumn Layout

Resizing the widget slightly forces the RowColumn to recompute its layout (see Figure 1-17), producing the same layout you get when you run your application.

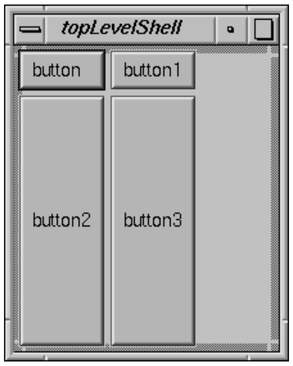


Figure 1-17 Corrected RowColumn Layout

Example: A Calculator

To demonstrate basic RapidApp use, this section describes how to create a simple calculator program that adds two integers. Figure 1-18 shows how the program looks when finished.

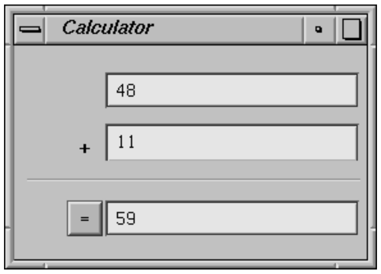


Figure 1-18 Completed Calculator Program

To create the calculator program:

1. Create an empty directory named *Calc* and start RapidApp.
2. Create a top-level window.

- Click the Simple Window icon from the Window palette.
- Position the pointer somewhere on the screen and click the left mouse button to place the window.
- In the Instance Name text field of the header area type "calcWindow" then press **`<Return>`**.
- In the resource named **title**, type "Calculator" then press **`<Return>`**.

3. Add a Bulletin Board container to the window by clicking the Bulletin Board icon in the Containers palette, positioning the pointer over the window, and clicking again.
4. Add an Text Field to the Bulletin Board.
   - Click the Text Field icon from the Controls palette.
   - Position the pointer over the Bulletin Board widget and click again.
   - Adjust the size and position of the widget, if necessary, to match the appearance shown in Figure 1-19.

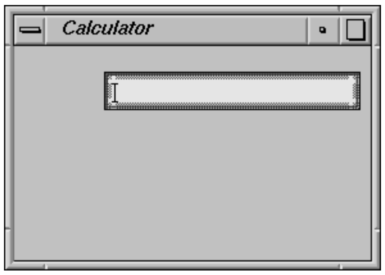


Figure 1-19 Initial Calculator Layout

5. Add a second Text Field below the first and place a label to the left of the second Text Field. Figure 1-20 shows the resulting layout.

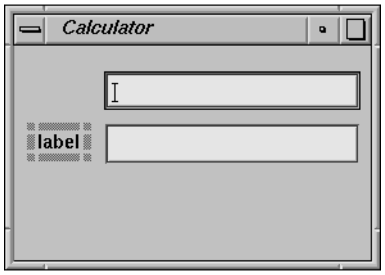


Figure 1-20 Second Text Field and Label

6. Complete the layout by adding a separator below the second Text Field, and a PushButton and third Text Field below the separator. Figure 1-21 shows the layout after all widgets have been placed.

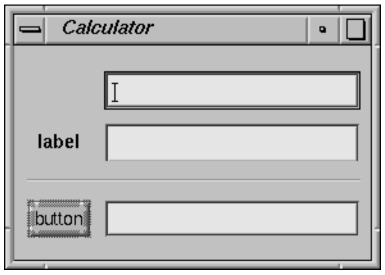
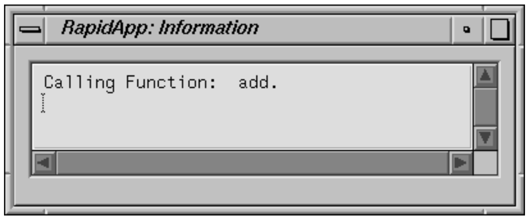


Figure 1-21 All Widgets In Place

7. Rename the top text field widget to “value1,” the second to “value2,” and the third to “result.” To do so, click each text field in turn, go the Instance Name field in the header area, and enter the new name.
8. Change the label on the Label widget to read “+”.
   - Click the label widget, and find the resource field named **labelString**.
   - Replace the text in that field with a “+” character.
   - Reposition the label if necessary.

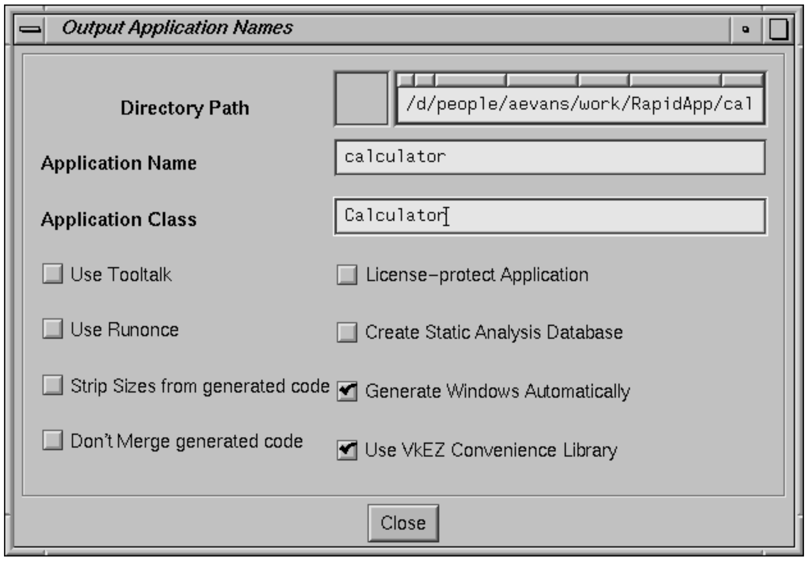
9. Change the label on the button widget to “=”.
   - Click the button widget, find the **labelString** resource field, and change the value to “=”.
   - Reposition the button if necessary.
10. Add a callback named “add” to the PushButton widget. With the button widget still selected, find the resource named **activateCallback** and type “add” then press **`<Return>`**. Notice that RapidApp automatically adds “()” after the function name. At this point, the interface should look like the window in Figure 1-18.
11. Test the interface by selecting the “Play Mode” option of the View menu. You can now type into the text fields, press the button, and so on. Notice as you press the button, an information window appears at the bottom of the screen (see Figure 1-22), reporting that the **add()** callback is being called.



```
Calling Function:  add.
```

Figure 1-22 RapidApp Information Window

12. Set code generation options.
    - Select “Build Mode” from the View Menu to go back to build mode.
    - Select “Application” from the Options menu.
    - In the dialog that appears, change the directory path to the *Calc* directory, if necessary.
    - Change the name field to “calculator” and the class name field to “Calculator, as shown in Figure 1-23. Make sure that the rest of the options are set as shown in Figure 1-23.
    - Click the *Close* button.



Directory Path: /d/people/aevans/work/RapidApp/cal

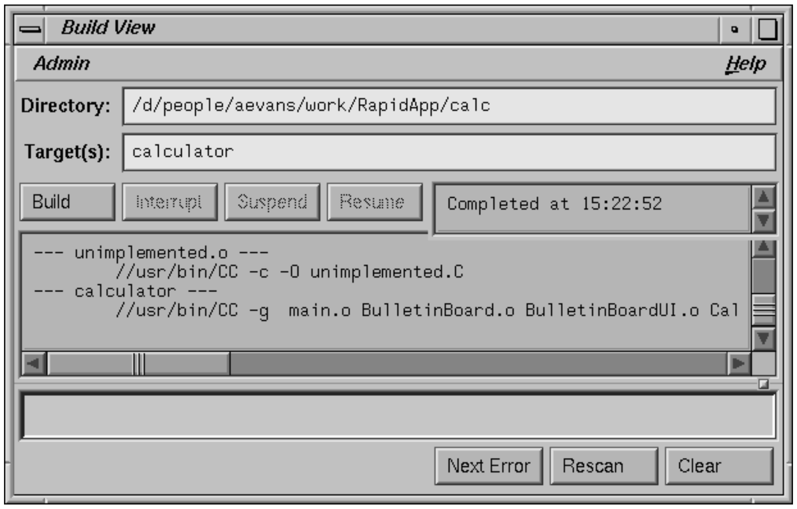
Application Name: calculator

Application Class: Calculator

- [ ] Use Tooltalk
- [ ] License–protect Application
- [ ] Use Runonce
- [ ] Create Static Analysis Database
- [ ] Strip Sizes from generated code
- [x] Generate Windows Automatically
- [ ] Don't Merge generated code
- [x] Use VkEZ Convenience Library

Figure 1-23 Application Options Dialog

13. Save the interface.
    - Select "Save" from the File menu.
    - RapidApp displays a dialog prompting you for a filename for the interface you have just created. Save the file as *calc.uil*. (The "uil" suffix stands for user interface language, and is a file format used by IRIS IM, as well as by many user interface tools. You should name your files with a ".uil" suffix.)
14. Generate code by selecting the "Generate C++" option of the Project menu. RapidApp displays a status window to report the files that it creates.
15. Build and run the program by selecting the "Run Application" item on the Project menu. The Developer Magic Build Manager (see Figure 1-24) appears and compiles the program. Once compiled, the program runs automatically.



```
/d/people/aevans/work/RapidApp/calc
calculator
Completed at 15:22:52
--- unimplemented.o ---
        //usr/bin/CC -c -O unimplemented.C
--- calculator ---
        //usr/bin/CC -g  main.o BulletinBoard.o BulletinBoardUI.o Cal
```

Figure 1-24 Building the Calculator Application

16. Add functionality.
    - Select the “Edit File” item from the Project menu pane.
    - When the file selection dialog appears, choose the file *BulletinBoard.C*.
    - When the text editor appears, scroll down until you locate the following section of code, which is the callback invoked when the user clicks the “=” pushbutton:

```
void BulletinBoard::add ( Widget w, XtPointer callData )
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct*) callData;

    //--- Comment out the following line when BulletinBoard::add is implemented:

    ::VkUnimplemented ( w, "BulletinBoard::add" );


    //--- Add application code for BulletinBoard::add here:


} // End BulletinBoard::add()
```

- Edit this function so that it appears as follows (your additions are shown in **bold**):

```
void BulletinBoard::add ( Widget w, XtPointer callData )
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct*) callData;

    //--- Comment out the following line when BulletinBoard::add is implemented:

    //::VkUnimplemented ( w, "BulletinBoard::add" );


    //--- Add application code for BulletinBoard::add here:

    int a = atoi(XmTextFieldGetString(_value1));
    int b = atoi(XmTextFieldGetString(_value2));
    XmTextFieldSetString(_result, (char *) VkFormat("%d", a + b));

} // End BulletinBoard::add()
```

The first two added lines call **XmTextFieldGetString()** to retrieve the contents of the top two text field widgets. Because this function retrieves a string, you must use **atoi()** to convert the string to an integer. Then **XmTextFieldSetString()** sets the resulting value in the result text field. **XmTextFieldSetString()** expects a string; this example uses the IRIS ViewKit convenience function **VkFormat()**, which works like **printf()** but returns a character string suitable for displaying in a text field widget. Notice that the widgets in this example are accessible in the BulletinBoard class as data members whose names are the names given in RapidApp but with a leading "_" added.

- Now scroll to the top of the file and add the header file for the **VkFormat()** function and <*stdlib.h*> for the **atoi()** function:

```
#include <Vk/VkFormat.h>
#include <stdlib.h>
```

17. Test the completed program.
    - Save the file and exit the editor.
    - Choose "Run Application" from the Project menu. The Build Manager appears again and builds the application. If you made any errors in typing in the changes, you can browse the errors using the Build Manager. Once compiled, the program runs automatically.

Figure 1-25 shows the completed application as it appears on the screen.

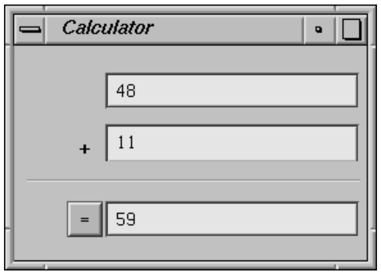


Figure 1-25 Working Calculator Program

- Try typing integer values into the test fields and pressing the "=" button.

18. You can package the product so that other users can install the calculator application using the Software Manager software installation tool (*swmgr*).

    - Go to the *Calc* directory and enter:

      ```
      % make image
      ```

This creates a complete installable image in a subdirectory named *images*.

EZ Convenience Functions

One problem for many developers new to IRIS IM is the amount of knowledge required to build working applications. Although RapidApp significantly reduces the knowledge needed to create application interfaces, you still need significant knowledge of IRIS IM to begin getting values from widgets, displaying data, or dynamically configuring widgets.

For example, the simple calculator program described in "Example: A Calculator" on page 24 requires that you know how to extract the contents of two text fields, convert the strings to integers, and add them back to the third text field. To do this simple operation, you must either know about the **XtSetValues()** / **XtGetValues()** functions and that the text widgets have an

XmNvalue resource, or you must know about the **XmTextFieldGetString()** / **XmTextFieldSetString()** functions.

Neither of these approaches is hard, but the fact that you have to know the interface for each type of widget can make seemingly simple tasks difficult. There are over 700 functions in IRIS IM, Xt, and Xlib, and although **XmTextFieldGetString()** and **XmTextFieldSetString()** are easy to use, you have to know they exist before you can actually use them.

The VkEZ package is a utility that makes it easier to perform simple operations in some cases. The package is not a general-purpose "widget wrapper" library and normally you shouldn't use it in production code—especially if you are concerned about the performance of your application. The VkEZ utility simply provides an easy-to-remember API for common operations. It is suitable for use in prototypes, demos, and applications in which performance isn't a concern. Instead of memorizing dozens or perhaps hundreds of IRIS IM functions, VkEZ requires that you remember only a few simple operations that you can apply to all widgets.

At its simplest, the VkEZ package provides a few simple operations you can apply to nearly any widget. You can use the "=", "<<", or "+=" operators to assign, or append data to a widget. The exact meaning of the operator varies with the widget but should normally "do the right thing." You can also retrieve the "value" of a widget simply by an implicit or explicit cast to the desired type. Again, the actual data returned depends on the widget. Retrieving the "String" of a Text widget yields the contents of the text field; retrieving the "String" of a List widget yields the text of the selected item. Retrieving an integer from a Scale widget gets the current value of the scale. Asking for the integer value of a text field returns the results of calling **atoi()** on the contents of the field.

To use a VkEZ operation, you must enclose the widget to be used in "EZ()", like this:

EZ(*widget*)

Then you can use the VkEZ operators to set and retrieve data from the widgets. For example, in the calculator example you can set the value of the *_result* text field to be the sum of the *_value1* and *_value2* widgets, like this:

```
EZ(_result) = EZ(_value1) + EZ(_value2);
```

You can also use the C++ "<<" operator to append data. For example, you can implement a more verbose form of the above example as follows:

```
EZ(_result) << "The result of " << EZ(_value1) << " + "
            << EZ(_value2) << " = "
            << EZ(_value1) + EZ(_value2);
```

If the *_value1* widget contains the string "10" and *_value2* contains "20," this places the following string in the *_result* text field:

```
The result of 10 + 20 = 30
```

Note: The VkEZ package is designed for quick prototypes and ease of learning. The implementation is inefficient and offers no real advantage over the IRIS IM API other than simplicity. Use the VkEZ utilities sparingly, and for production-quality programs, plan to replace all uses with the more direct mechanisms supported by IRIS IM. When you are ready to replace the EZ functions with production code, you should be able to find all occurrences of "EZ" quite easily in your editor.

For more detailed information about the widgets and operations supported, see Appendix E, "VkEZ Reference."

Examples Using the EZ Functions

The VkEZ package relies on a simple model that assumes you want to do the most obvious operation for a given widget. For example, assume you want to increment a Dial widget, represented by the data member *_dial*, by 10 each time a particular function is called. In the function, you can simply write:

```
EZ(_dial) += 10;
```

Suppose you want to tie two dials, *_dial1* and *_dial2*, together so that *_dial2* always displays 1/2 the value of *_dial1*. You can do so by including the following code to the function invoked when *_dial1* changes value:

```
EZ(_dial2) = EZ(_dial1) / 2;
```

List widgets can be difficult to work with, and EZ provides an easy way to set, add, or retrieve the contents of a list. For example, you can display a list of strings in a List widget like this:

```
EZ(_list) = "red, green, blue";
```

You can add colors later with:

```
EZ(_list) += "yellow, orange";
```

or:

```
EZ(_list) << "yellow, orange";
```

Support for Widget Resources

The VkEZ package also provides access to several common IRIS IM resources. For example, you can set or get the width, height, or position of a widget. The following code segment displays a string in a text widget named *_text* that reports the width of a *_button* widget:

```
EZ(_text) = "The width of the button is "
             << EZ(_button).width << " pixels";
```

You can set the width of a label widget, *_label*, to be the same as another, *_longlabel*, with:

```
EZ(_label).width = EZ(_longlabel).width;
```

You can set a color using:

```
EZ(_label).foreground = "blue";
```

You can even use colors defined by schemes as shown in this example:

```
EZ(_label).background =
                     "Sgi_DYNAMIC AlternateBackgroundColor1";
```

Chapter 2

Creating Applications with RapidApp

This chapter tell you how to develop applications effectively using RapidApp.

Chapter 2

Creating Applications With RapidApp

This chapter describes the process of developing an application using RapidApp:

- "RapidApp Development Model" on page 37 discusses the general development model supported by RapidApp.
- "RapidApp Development Cycle" on page 49 describes the typical development cycle for creating an application with RapidApp.

RapidApp Development Model

RapidApp can significantly simplify the process of creating an application, not only in creating the interface for your application but also in managing the development of your application from prototype to finished product. To provide this support, RapidApp assumes a certain application development model. Although you can "go around" RapidApp and force it into a model that it isn't intended to support, you won't derive the full benefits of using RapidApp.

This chapter describes the general development model best supported by RapidApp and provides tips for getting the most out of RapidApp. The key features of RapidApp's development model, which are described in following sections, are:

- Creating applications from object-oriented components
- Integrating with the Indigo Magic Desktop environment
- Separating interface code from functional code
- Managing code evolution
- Integrating with the Developer Magic / ProDev WorkShop tools
- Packaging applications for installation

Object-Oriented Components

RapidApp supports the object-oriented architecture defined by the IRIS ViewKit class library. The fundamental building blocks in the IRIS ViewKit library are *components*, which are C++ classes that encapsulate one or more widgets and define the behavior of the overall components.

As an example, consider a simple spreadsheet. You can create a spreadsheet interface using IRIS IM text field widgets for the individual cells and an IRIS IM container widget to display the text fields in a grid. An IRIS ViewKit spreadsheet component can be a C++ class containing not only these widgets, but also the code implementing the spreadsheet functionality. In your application, you can then instantiate a spreadsheet component and interact with it by calling various member functions. If the spreadsheet component is properly designed, you can reuse it in applications needing a spreadsheet. Furthermore, you can extend the functionality of the basic spreadsheet component by creating subclasses as needed. For example, you can implement a general-purpose spreadsheet component, then create subclasses in other applications adding special financial or scientific functions.

RapidApp allows you to define a component consisting of any collection of widgets—or even of a single widget. When you do this, RapidApp places in a C++ class the widgets in the component along with the callbacks and other resources for those widgets. You can specify the name of the class as well as the name of the files in which RapidApp places the code. When you create a component, RapidApp also adds it to a special “User Defined Components” palette. You can select the component from this palette and add it to your interface just like any other interface element.

You can nest components, building more and more complex components from simpler elements. Ideally, you can even view your entire application as a component, allowing you to incorporate it later within a larger application or suite of applications. RapidApp encourages this approach by automatically encapsulating the entire contents of each top-level window in your application within separate classes if they aren’t already a components; each top-level window of your application then simply creates instances of these classes. You can just as simply instantiate these top-level classes as part of another application.

Tip: A good technique for developing applications in RapidApp is to create collections of small components. It's easiest to get the layouts you want by working with smaller, simpler pieces that are "frozen" as self-contained objects. Then you can use these components to construct more complex objects.

Integration With the Indigo Magic Desktop Environment

RapidApp is designed to produce applications that integrate with the Indigo Magic Desktop environment. Some features automatically included in applications developed with RapidApp include:

- A generic Desktop icon for your application along with a generic File-Type Rule (FTR) file (see Chapter 2, "Icons," in the *Indigo Magic User Interface Guidelines* and Chapter 11, "Creating Desktop Icons: An Overview," in the *Indigo Magic Desktop Integration Guide* for instructions on customizing the look and behavior of your application's Desktop icons)
- The Indigo Magic look for widgets (see Chapter 3, "Windows in the Indigo Magic Environment," in the *Indigo Magic User Interface Guidelines* and Chapter 2, "Getting the Indigo Magic Look," in the *Indigo Magic Desktop Integration Guide* for information on the Indigo Magic look)
- Support for font and color schemes (see Chapter 3, "Windows in the Indigo Magic Environment," in the *Indigo Magic User Interface Guidelines* and Chapter 3, "Using Schemes," in the *Indigo Magic Desktop Integration Guide* for information on schemes)
- Proper window decorations and window menu entries for each window type (see Chapter 3, "Windows in the Indigo Magic Environment," in the *Indigo Magic User Interface Guidelines* and Chapter 5, "Window, Session, and Desk Management," in the *Indigo Magic Desktop Integration Guide* for information on window decorations and window menu entries)
- Standard menu bar entries with keyboard accelerators when you use the VkWindow interface element (see Chapter 8, "Menus," in the *Indigo Magic User Interface Guidelines* for information on standard menu bar entries)

To take advantage of these features fully, don't try to "go around" RapidApp to implement them yourself. For example, don't set the color or font of an interface element directly; instead, let the schemes mechanism assign the fonts and colors to your application based on the user's selected scheme. To prevent you from accidentally doing so, RapidApp doesn't allow you to set fonts or colors directly. You can still edit the source code or resource file to override the default color or font, but if you do so, use the special symbolic scheme colors and fonts as described in Chapter 3, "Using Schemes," in the *Indigo Magic Desktop Integration Guide*.

Code Management

RapidApp generates all the files needed to build your application. However, although RapidApp can generate a significant portion of your application's code, you must still write some parts using a text editor. This introduces the challenge of keeping RapidApp and your text editor from interfering with each other; you don't want to lose changes made in one when you make changes using the other. RapidApp addresses this challenge in two ways: object-oriented design and code merging. This section describes the files that RapidApp generates, how they work together, and how RapidApp merges changes that you make into the code that it generates.

Code Generation

To understand how all of the files that RapidApp generates work together, consider a very simple example—the calculator program described in "Example: A Calculator" on page 24. To demonstrate more of the RapidApp code generation features, assume that you've encapsulated the calculator interface into a component called **Calculator**. "Creating Components" on page 89 demonstrates how to do this.

To generate the C++ code for your application, select "Generate C++" from the Project menu. In the case of the calculator program, RapidApp generates the following files and directory:

```
main.C
CalcWindowMainWindow.h
CalcWindowMainWindow.C
CalculatorUI.h
CalculatorUI.C
Calculator.h
Calculator.C
Calculator
Makefile
calculator.idb
calculator.spec
desktop.ftr
icon.fti
unimplemented.C
.buildersource/ (directory)
```

These files fall into four basic categories: the program driver, a top-level window class, components, and configuration or support files:

Driver

All programs have a *main.C* file that instantiates a **VkApp** object and one or more top-level windows. See Chapter 3, "The ViewKit Application Class," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkApp** class. You don't have to use the generated *main.C*, but there is little reason to have anything different.

Note: If you want to handle ToolTalk™ messages in your application, you need to select "Application" from the Options menu and toggle on the use ToolTalk option. This causes RapidApp to instantiate a **VkMsgApp** object instead of a **VkApp** object. See Appendix A, "ViewKit Interprocess Message Facility," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkMsgApp** class and the IRIS ViewKit support for ToolTalk.

Top-level window

In this case, RapidApp generates the **CalcWindowMainWindow** class in two files: *CalcWindowMainWindow.h* and *CalcWindowMainWindow.C*. It implements top-level windows as subclasses of either **VkSimpleWindow** or **VkWindow**. See Chapter 4, "ViewKit Windows," in the *IRIS ViewKit Programmer's Guide* for more information on these classes. In this example, the **CalcWindowMainWindow** class simply instantiates a **Calculator** object.

Component files

The only component in this example is the **Calculator** class. Components are subclassed from the IRIS ViewKit **VkComponent** class. See Chapter 2, "Components," in the

IRIS ViewKit Programmer's Guide for more information on the **VkComponent** class. As discussed in "Object-Oriented Design," RapidApp splits the **Calculator** class into **CalculatorUI** and **Calculator**; the user interface designed in RapidApp is implemented in **CalculatorUI**, while the **Calculator** class is mostly empty. For best results, add application code to only the *Calculator.C* and *Calculator.h* files.

Support files — These include a *Makefile*, the application resources file (*Calculator*), files used for Software Packager (*calculator.idb* and *calculator.spec*), and files for your application's Desktop icon (*desktop.ftr* and *icon.fti*). A few of these files are of special interest. For example, the *.buildersource* directory supports RapidApp code merge features, as described in "Code Merging" on page 43. It is also important that you not move or rename these files. Another file of interest is the *unimplemented.C* file, which contains the function **VkUnimplemented()** described in "Debugging and Interactively Adding Functionality" on page 47.

Object-Oriented Design

To minimize conflicts when it generates code for a component, RapidApp generates two separate C++ classes. One class, which usually has the suffix "UI" appended to the class name, contains all the code needed to generate the user interface, including creating components and IRIS IM widgets, registering callbacks, and so on. The second generated class is a subclass of the "UI" class and contains the code that implements the actual functionality of the component.

Separating the user interface code from the functional code allows RapidApp to "own" the UI class. Generally, you should make changes only to the derived class. The UI base class declares all widgets and components as protected data members so that you can access and manipulate them freely in the derived class. RapidApp implements widget callback functions in the base UI class. Each callback corresponds to a virtual function declared initially in the base class but overridden in the derived class. You can use any text editor to complete the bodies of the derived class's virtual functions.

With this separation of interface and functional code, when you make changes to a component’s interface, RapidApp can update the UI class without affecting the subclass containing the functional code. In fact you can completely redesign a component’s interface, but as long as you retain the same callback functions, you might require only minimal changes to the functional code in the derived class. (You might need to change some widget access code, for example, if you replace a radio box and toggle buttons with an option menu.)

Tip: In general, to minimize difficulties in updating code, add and change code in only the derived classes (that is, those classes without the “UI” suffix).

Code Merging

When you make changes using RapidApp and then generate new code, RapidApp attempts to merge each file it manages. Each time you generate code, RapidApp performs the following steps for each file (these steps use *Makefile* as an example):

1. RapidApp writes the newly generated *Makefile* to a different name, *.Makefile.N*.
2. If *Makefile* doesn’t exist, RapidApp moves *.Makefile.N* to *Makefile*. RapidApp also saves *Makefile* to a hidden subdirectory in your product directory, *.buildersource*. RapidApp then terminates the merge process.
3. If *Makefile* exists, RapidApp compares *.Makefile.N* to *Makefile*. If there are no differences, RapidApp removes *.Makefile.N* and terminates the merge process.
4. If there are differences between the newly generated file and the current *Makefile*, RapidApp compares *.Makefile.N* to version it generated previously (which it stored in *.buildersource*). If there are no differences between those two versions, then you must have added the changes to the current *Makefile* by hand, so RapidApp removes *.Makefile.N* and terminates the merge process.
5. If there are differences in all three files, RapidApp initiates a three-way merge (see the *merge*(1) reference page), which treats the last known file generated by RapidApp as an ancestor and compares your changes (if any) to *Makefile* to those found in *Makefile.N* and attempts to resolve the differences. If the differences were resolved successfully, the merged

changes are made to *Makefile*. *Makefile.N* becomes the new ancestor, and is saved in the *.buildersource* directory to be used in future merges. RapidApp also saves the original file in a *.backup* subdirectory as *Makefile.<#>* (where <#> is a generated number) as a guard against any possible failure of the merge.

6. If the merge process couldn't resolve all differences, RapidApp offers you the option to manually merge the files, to discard the current *Makefile*, or to keep the current *Makefile*. In all three cases, RapidApp copies the original file to *Makefile.<#>*, where <#> is the highest number not currently in use.
   - If you choose to merge, RapidApp invokes an interactive merge tool that visually shows the areas that are in conflict and offers you a chance to manually resolve the differences.
   - If you choose to discard the current *Makefile*, RapidApp overwrites the current file with the newly generated file.
   - If you choose to keep the current *Makefile*, RapidApp moves the generated file to *Makefile.New* and leaves the current file untouched.

In addition to these steps, RapidApp takes some precautions to address missing builder files. Each time RapidApp generates code, it copies the builder's *uil* file used to generate the code to a checkpoint file, *.checkpoint.uil*. If this file exists already, RapidApp first copies the old file to *.checkpoint.prev.uil*. Then, if RapidApp can't locate a file in the *.buildersource* subdirectory during the code generation process, it uses the previous *uil* file to repopulate the *.buildersource* subdirectory with the files that are missing before proceeding with the code generation and merging process.

Note: For the most part, you should not need to be aware of the code merging process. These details are provided to help you understand the underlying mechanisms both to promote confidence and to help you recover from difficulties in case anything should go wrong.

The code merging process is usually successful if you follow a few simple rules:

- Do not arbitrarily reformat any generated files.
- Try to limit your changes to the areas marked by comments like:

```
//--- End generated code section
```

These comments indicate which areas are owned by RapidApp and which are free for you to modify. They also help the merge program stay on track. You can make changes anywhere you like, but you have less chance of making overlapping changes later when using RapidApp if you limit changes to the designated areas.

- Although you can modify any file (RapidApp merges all files), try to limit changes to source code to the derived classes whenever possible. There should be little reason to modify the base UI classes because anything you need to do can be done in the derived class. You can even add, remove, or manipulate widgets in the derived class member functions.
- Use only RapidApp to make changes to the interface

  Note: RapidApp doesn't reflect any changes you make to your program's interface by directly editing the code. For example, if you cut the creation of a widget directly from the code, the widget still appears the next time you run RapidApp. This is because RapidApp doesn't read the source code but instead reads a separate file used to save a description of the interface. However, because of the merging strategy, changes that you make directly to the interface source code aren't lost; RapidApp simply doesn't display your changes.
- Use the hooks explicitly provided for extensions. For example, the *Makefile* defines a USERFILES variable, which is meant for adding files that are created outside the builder.

RapidApp can do a lot for you. If you reorder functions, reformat code, or otherwise modify a file to the point that RapidApp can't identify the original structure programmatically, your ability to continue to use RapidApp to modify the file is severely limited.

Integration With ProDev WorkShop for Building and Debugging

An important feature of RapidApp is that it integrates with the Developer Magic ProDev WorkShop tools, providing a rich environment for building and debugging applications. You access the ProDev WorkShop tools through the RapidApp Project menu. The following sections give an overview of the tools available.

For more information about the ProDev WorkShop tools, see “ProDev WorkShop and MegaDev Overview” in *Developer Magic: ProDev WorkShop and MegaDev Overview*, which provides an introduction to the tools.

Editing Files

You can start the SourceView editor by selecting “Edit Files” from the Project menu. RapidApp then displays a file selection dialog prompting you for the file to edit. Once you select the file, RapidApp starts the SourceView editor.

If you want, you can specify a different text editor for RapidApp to invoke. To do so, set the $WINEDITOR environment variable to the editor you want to invoke. Then in RapidApp, select “RapidApp Preferences” from the Options menu and toggle on the “Use $WINEDITOR to edit files” option.

Compiling

You can compile your applications by selecting “Build Application” from the Project menu. This launches and starts the Developer Magic Build Manager. If you are currently using the debugger, the executable is automatically detached from the debugger and reattached when the compilation is completed.

Browsing Source

You can configure the *Makefile* created by RapidApp to automatically create a static analysis fileset and database for all generated files. To do so:

1. Select “Application” from the Options menu to display the Output Application Names dialog.
2. Toggle on the Create Static Analysis Database checkbox and close the dialog.
3. Select “Generate C++” from the project menu to update your files, include the *Makefile*.

The next time you build your application, RapidApp creates the static analysis files. The files are kept in a subdirectory named *<DirectoryName>.cvdb*, where *<DirectoryName>* is the name of your project directory, so they do not clutter the work area. After creating the static analysis files, you can select “Browse Source” from the Project menu to launch the Static Analyzer.

Note: If you add new files outside RapidApp, you need to add them to the fileset file manually.

Debugging and Interactively Adding Functionality

You can launch the Debugger by choosing “Debug Application” from the Project menu. If the program is not up-to-date, RapidApp automatically invokes the Build Manager to update the executable.

Besides the obvious uses of the Debugger for finding and fixing bugs, you can also use the Fix and Continue tool from the Debugger to interactively add functionality to your program. To do so:

1. Run the program.
2. Click buttons, select menu items, and otherwise exercise your program’s interface. As you hit each unimplemented function, the Debugger stops in **VkUnimplemented()**.
3. Because **VkUnimplemented()** is called by the virtual function you really want to modify, click the *Return* button in the Debugger to go up one level.
4. Choose “Edit” from the Fix+Continue menu in the Debugger. The body of the code changes color to indicate the editable region.
5. Type in your changes.
6. Select “Parse And Load” from the Fix+Continue menu.
7. Click the *Continue* button in the Debugger to resume running your program.
8. Test the behavior of the code you added. When you are satisfied, comment out the call to **VkUnimplemented()** from your function so the debugger no longer stops in this callback.

9. Repeat the procedures from step 2 to continue to add functionality to your program.
10. Once you are finished adding functionality, select “Save File+Fixes As” from the Debugger’s Fix+Continue menu to save your changes.

Integration With Software Packager for Creating Installable Images

RapidApp automatically generates the files required to create an image that users can install with the Software Manager installation tool (*swmgr*). You can generate a default image by going to the directory that contains your source and entering:

```
% make image
```

This creates a subdirectory named *images* containing the installable image. Remember to include all of the files in this directory in your distribution.

The default image created by RapidApp consists of a minimal set of application files: the executable, the default resources file, the Desktop icon, and the FTR file. You might want to customize the image to include other files, divide the product into base and optional subproducts, or include commands to be executed after installation. For example, if you have reference pages (man pages) for your product, you should edit your images to include them.

To edit your product’s images, select “Edit Installation” from the Project menu. Doing so launches Software Packager, a graphical tool for creating and editing installable images. For complete instructions for using Software Packager, consult the *Software Packager User’s Guide*; Chapter 1, “Packaging Software for Installation: An Overview,” provides an overview of the tool.

RapidApp Development Cycle

The RapidApp development cycle typically consists of the following steps:

1. Use RapidApp to create a graphical user interface for your application.
2. Run your prototype interface under the Debugger and use the Fix and Continue tool to create prototype functional code. You can also add functionality using the SourceView tool or external text editors. To generate a working prototype quickly, you might want to use the VkEZ convenience functions. See “ProDev WorkShop and MegaDev Overview” in *Developer Magic: ProDev WorkShop and MegaDev Overview* for more information about the ProDev WorkShop tools.
3. Use RapidApp to refine your user interface based on testing and feedback.
4. Develop and test any external (that is, non-interface) functionality required by your application. At this point, you might also decide to encapsulate portions of your interface as self-contained components.
5. If you used VkEZ convenience functions, replace them with production code.
6. Perform final testing and make any necessary revisions.
7. Use IconSmith to create a custom Desktop icon for your application and edit the FTR file to customize the behavior of your application’s icon. See Chapter 2, “Icons,” in the *Indigo Magic User Interface Guidelines* for guidelines and Chapter 11, “Creating Desktop Icons: An Overview,” in the *Indigo Magic Desktop Integration Guide* for instructions on customizing the look and behavior of your application’s Desktop icons.
8. Create a minimized window icon to represent your application when iconified. See Chapter 3, “Windows in the Indigo Magic Environment,” in the *Indigo Magic User Interface Guidelines* for guidelines and Chapter 6, “Customizing Your Application’s Minimized Windows,” in the *Indigo Magic Desktop Integration Guide* for instructions on creating minimized window icons.
9. Use Software Packager to customize your application’s installable images. For complete instructions for using Software Packager, consult the *Software Packager User’s Guide*; Chapter 1, “Packaging Software for Installation: An Overview,” provides an overview of the tool.

Chapter 3

Building Interfaces with RapidApp

This chapter provides tips for selecting and using interface elements for your application.

Chapter 3

Building Interfaces With RapidApp

This chapter describes how to select and use interface elements to create your application's interface:

- "Choosing and Using Windows," gives you tips on how to use top-level windows.
- "Using Containers," discusses the advantages and limitations of the different containers available.
- "Creating and Editing Menus," gives instructions for creating menu bars and options menus.
- "Creating, Editing, and Manipulating Components," tells you how to create and use self-contained interface components.

This chapter focuses on choosing appropriate interface elements for your interface rather than discussing the features of each element in detail. For detailed information about the resources available for each interface element, see Appendix B, "RapidApp Reference."

Choosing and Using Windows

Most often, the first step in creating an interface with RapidApp is to select an appropriate window. This is not the case if you're using RapidApp to create only self-contained components. In that case, the window you use to hold your component as you build it is irrelevant; you're interested in the component the window holds.

All of the windows are available on the Windows palette. As shown in Figure 3-1, RapidApp provides a choice of three top-level windows: a Simple Window, a VkWindow, and a Dialog Window. This section describes the features of these windows and when it's appropriate to use each type.

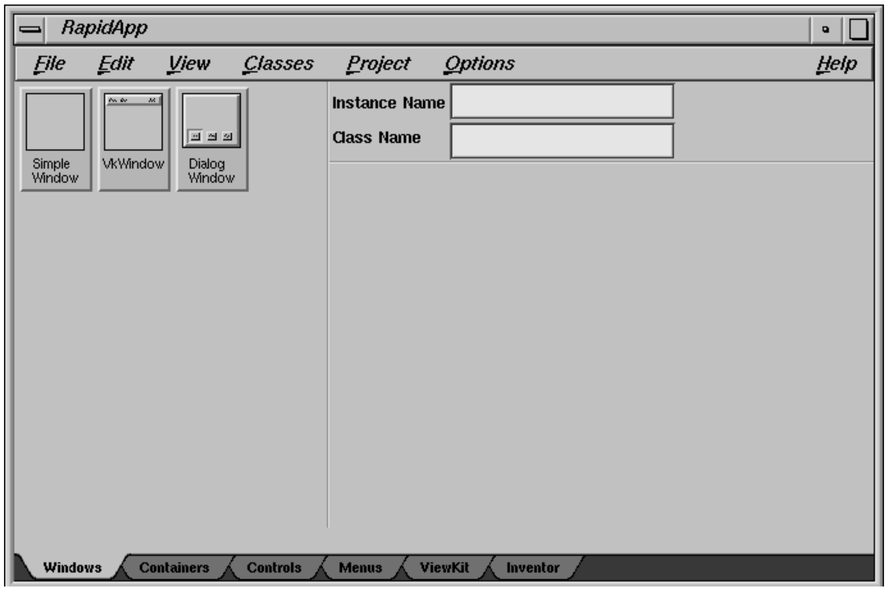


Figure 3-1 The RapidApp Windows Palette

Note: You can also create containers without first creating a top-level window. If you do so, RapidApp automatically provides a top-level shell for the container. Then when you generate code, RapidApp automatically generates the equivalent of a Simple Window for that shell.

For each type of window, RapidApp automatically provides appropriate Indigo Magic window decorations and window menu entries. See Chapter 3, “Windows in the Indigo Magic Environment,” in the *Indigo Magic User Interface Guidelines* and Chapter 5, “Window, Session, and Desk Management,” in the *Indigo Magic Desktop Integration Guide* for information on window decorations and window menu entries.

Simple Windows

As its name implies, a Simple Window is the simplest top-level window. A Simple Window has no menu bar and can contain only one child element. If you want a menu bar for the window, create a VkWindow instead of a Simple Window. The child element that you place in a Simple Window is typically either a container widget or a complex component. You can use Simple Windows as *main windows*, but typically they're more appropriate as *co-primary windows*. See "Main Primary and Co-Primary Windows" on page 58 for more information on main and co-primary windows.

When RapidApp generates code for a Simple Window, it creates it as a subclass of the IRIS ViewKit **VkSimpleWindow** class. (See Chapter 4, "ViewKit Windows," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkSimpleWindow** class.) Furthermore, if the child of the Simple Window isn't a component (that is, a C++ class), RapidApp automatically encapsulates that child and its contents within a subclass of **VkComponent**. (See Chapter 2, "Components," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkComponent** class.) The Simple Window then simply creates an instance of this class.

VkWindows

A VkWindow supports far more functionality than a Simple Window. Although it, like the Simple Window, can contain only one child element, a VkWindow includes a menu bar with many of the standard menu bar entries complete with keyboard accelerators (see Chapter 8, "Menus," in the *Indigo Magic User Interface Guidelines* for information on standard menu bar entries). You typically use VkWindows as *main windows*, but you can use them as *co-primary windows* as well. See "Main Primary and Co-Primary Windows" on page 58 for more information on main and co-primary windows.

Figure 3-2 shows the default configuration of the VkWindow component.

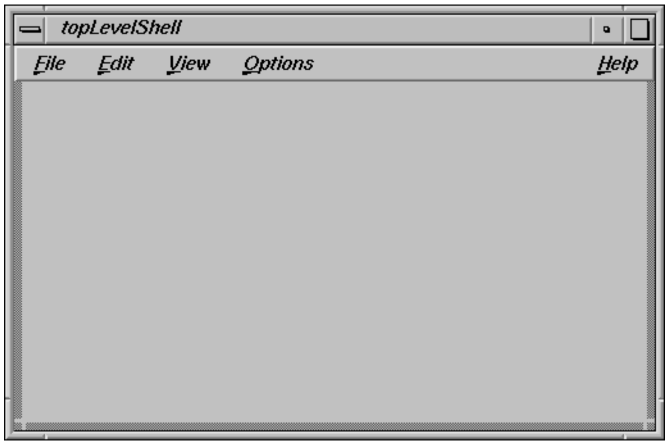


Figure 3-2 Default Configuration of VkWindow Component

When RapidApp generates code for a VkWindow, it creates it as a subclass of the IRIS ViewKit **VkWindow** class. (See Chapter 4, "ViewKit Windows," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkWindow** class.) Furthermore, if the child of the VkWindow isn't a component (that is, a C++ class), RapidApp automatically encapsulates that child and its contents within a subclass of **VkComponent**. (See Chapter 2, "Components," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkComponent** class.) The VkWindow then simply creates an instance of this class.

For each menu item in the menu bar for which you've defined an **activateCallback** function, RapidApp adds a member function of the same name to the VkWindow's child component or generated class. You can then add the functional code to the functions to define behavior for the menu items. RapidApp doesn't add member functions to the child for those menu items for which you haven't defined an **activateCallback** function.

Furthermore, for the default items on the File and Edit menus, RapidApp implements some functionality automatically. For example, RapidApp generates code for the "Open" selection of the File menu to display a file selection dialog. You need only take the filename returned by the dialog and perform an open operation appropriate for your application. Table 3-1 summarizes the actions and the functions added for the default items on the File and Edit menus.

Table 3-1 Default Actions of Standard VkWindow Menu Items

Menu	Selection	Function Added to Child Component or Class	Description
File	New	**newFile()**	The child creates a new, empty file.
	Open	**openFile(const char *)**	The VkWindow automatically displays a file selection dialog and, if the user selects a file, passes that filename to the child as an argument to the **openFile()** function. The child opens the given file.
	Save	**save()**	The child saves its current state to the current file.
	Save As	**saveas(const char *)**	The VkWindow automatically displays a file selection dialog and, if the user selects a file, passes that file name to the child as an argument to the **saveas()** function. The child saves its current state to the specified file.
	Print	**print(const char *)**	The child prints its contents. Currently, the argument to **print()** is unused.
	Close		The VkWindow deletes itself. To change this behavior, edit the **close()** function of the **VkWindow** subclass.
	Exit		The VkWindow calls **VkApp:quitYourself()** to exit the application. (See Chapter 3, "The ViewKit Application Class," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkApp** class.) To change this behavior, edit the **quit()** function of the **VkWindow** subclass.
Edit	Undo		The VkWindow automatically invokes the undo functionality provided by the IRIS ViewKit **VkMenuUndoManager** class. (See Chapter 6, "ViewKit Undo Management and Command Classes," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkMenuUndoManager** class.) You can't override this behavior; if you don't want to support undo in your application, remove this menu item.
	Cut	**cut()**	The child cuts its current selection to the clipboard. See Chapter 5, "Data Exchange on the Indigo Magic Desktop," in the *Indigo Magic User Interface Guidelines* for guidelines and Chapter 7, "Interapplication Data Exchange," in the *Indigo Magic Desktop Integration Guide* for instructions on implementing cut and paste in your application.

Table 3-1 (continued) Default Actions of Standard VkWindow Menu Items

Menu	Selection	Function Added to Child Component or Class	Description
	Copy	**copy()**	The child copies its current selection to the clipboard. See Chapter 5, “Data Exchange on the Indigo Magic Desktop,” in the *Indigo Magic User Interface Guidelines* for guidelines and Chapter 7, “Interapplication Data Exchange,” in the *Indigo Magic Desktop Integration Guide* for instructions on implementing cut and paste in your application.
	Paste	**paste()**	The child retrieves the contents of the clipboard and inserts it as appropriate. See Chapter 5, “Data Exchange on the Indigo Magic Desktop,” in the *Indigo Magic User Interface Guidelines* for guidelines and Chapter 7, “Interapplication Data Exchange,” in the *Indigo Magic Desktop Integration Guide* for instructions on implementing cut and paste in your application.

You can add, edit, and remove menu panes and menu items if you want. For more information on manipulating menus in RapidApp, see “Creating and Editing Menus” on page 84.

Note: Don’t remove or edit the Help menu. The **VkWindow** class automatically creates a standard Help menu that interfaces with the Silicon Graphics help system; RapidApp ignores any changes that you make to the Help menu. For more information on the standard Help menu, see Chapter 5, “Creating Menus With ViewKit,” in the *IRIS ViewKit Programmer’s Guide*.

Main Primary and Co-Primary Windows

Chapter 3, “Windows in the Indigo Magic Environment,” of the *Indigo Magic User Interface Guidelines* describes two types of primary windows recommend for use in Indigo Magic Desktop applications: *main primary windows* and *co-primary windows*:

- A main primary window serves as the application’s main controlling window. It’s used to view or manipulate data, get access to other windows within the application, and kill the process when users quit. You should have only one main primary window per application.

- A co-primary window is used for major data manipulation or viewing of data outside of the main window. Co-primary windows are often used as “auxiliary” windows and are not displayed automatically on starting the application.

RapidApp allows you to set the type of a Simple Window or a VkWindow with the **coprimaryWindow** resource, as shown in Figure 3-3.

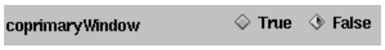


Figure 3-3 Setting the Window Type

Chapter 3 of the *Indigo Magic User Interface Guidelines* recommends different entries in the window menu (that is, the menu in the title bar added by the window manager) based on the type of window. RapidApp automatically generates the necessary code to configure a window based on the value of the **coprimaryWindow** resource that you select.

RapidApp automatically instantiates and displays all main windows in *main.C*. However, it doesn’t create instances of or display co-primary windows in your application. You need to instantiate co-primary windows explicitly and use the **show()** member function to display them when appropriate. For example, if you display a co-primary window based on an action in another component, you can declare the co-primary window as a protected data member of the component:

```
protected:
  CoprimaryMainWindow * _coprimary;
```

Then instantiate the co-primary window in the component’s constructor:

```
_coprimary = new CoprimaryMainWindow("coPrimary");
```

When you need to display the co-primary window, call its **show()** member function:

```
_coprimary->show();
```

See Chapter 4, “ViewKit Windows,” in the *IRIS ViewKit Programmer’s Guide* for more information on manipulating windows using the window class member functions.

Dialog Windows

Typically, you don't need to create basic Dialog Windows using RapidApp. By comparison to other parts of your interface, there is little to customize for most dialogs. You simply need a way to specify a message and title, post and dismiss the dialog, and perhaps retrieve a value. Furthermore, dialogs are rarely posted as a result of specific user interaction such as clicking a button; instead, they are often a result of error conditions or other program states.

For most dialogs in your application, use the standard dialog posting mechanism provided by IRIS ViewKit. IRIS ViewKit implements a complete dialog management system including:

- caching and reusing dialogs to improve application performance
- single function mechanisms for posting dialogs
- ability to post any dialog in non-blocking, non-modal mode; modal mode; and two blocking modes
- positioning in multi-window applications
- posting of dialogs even when windows are iconified, if desired
- correct handling of dialog references when widgets are destroyed

The IRIS ViewKit dialog mechanism handles all standard dialog types including information, warning, error, busy, question, prompt, file selection, and preference dialogs. For more information on the IRIS ViewKit dialog mechanism, see Chapter 7, "Using Dialogs in ViewKit," in the *IRIS ViewKit Programmer's Guide.*

Occasionally, you might need to create a custom dialog not implemented in IRIS ViewKit. The Dialog Window element on the Windows palette allows you to create a custom dialog that integrates with the IRIS ViewKit dialog mechanism.

The Dialog Window is the basis for your custom dialogs. Figure 3-4 shows the default configuration of a Dialog Window.

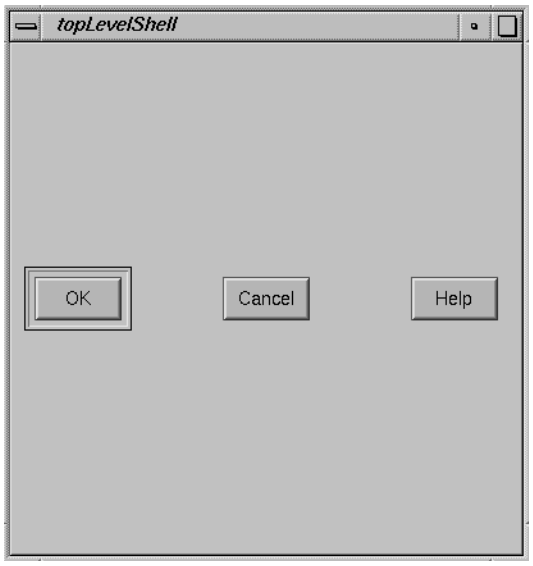


Figure 3-4 Default Configuration of Dialog Window

Besides the standard dialog buttons, a Dialog Window can contain only one other child element. The child element that you place in a Dialog Window is typically either a container widget or a complex component.

When RapidApp generates code for a Dialog Window, it creates the dialog as a subclass of the IRIS ViewKit **VkGenericDialog** class. (See Chapter 7, "Using Dialogs in ViewKit," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkGenericDialog** class.) Furthermore, if the child of the Dialog Window isn't a component (that is, a C++ class), RapidApp automatically encapsulates that child and its contents within a subclass of **VkComponent**. (See Chapter 2, "Components," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkComponent** class.) The Dialog Window then simply creates an instance of this class.

Note: RapidApp generates both a UI and a functional subclass for the child of your custom dialog. Typically, you should edit only the subclass.

Figure 3-5 shows an example of a custom dialog. In this case, when you generate code, the container widget containing the scale and label is encapsulated into a subclass of **VkComponent**. Alternatively, you could select the container widget and explicitly create a component, **VolumeControl** for example, before generating code.

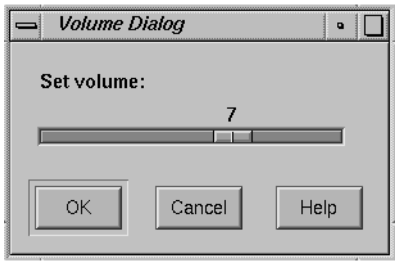


Figure 3-5 Example of a Custom Dialog

When RapidApp generates the code for the dialog’s child class, it adds four member functions to it: **ok()**, **cancel()**, **apply()**, and **help()**. These functions are called when the user clicks the corresponding button. You can add code to these functions to perform whatever tasks you require. In the case of the custom dialog shown in Figure 3-5, the **VolumeControl::ok()** function can store the current value of the scale widget into a data member; the **VolumeControl::cancel()** function can restore the scale to the previously stored value.

Because a custom dialog is a subclass of **VkGenericDialog**, you post, dismiss, and set dialog titles and button labels the same way as for any other IRIS ViewKit dialog. For example, if the name of the dialog class for the dialog shown in Figure 3-5 is **VolumeDialog**, you create an instance of the dialog in your program with:

```
VolumeDialog _volumeDialog = new VolumeDialog();
```

Then you post this dialog with a call such as:

```
_volumeDialog->post();
```

See Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide* for more information on manipulating dialogs in IRIS ViewKit.

If you want to retrieve values set in the dialog or otherwise manipulate the dialog, create these access functions in both the dialog class and the child class. The dialog class should simply call the corresponding function in the child class. For example, assume that you want to be able to retrieve the last value of the scale in the dialog shown in Figure 3-5. Assume also that the dialog's child class, **VolumeControl**, stores the value in a private data member, *_scaleValue*. First, add the following function to the **VolumeDialog** class:

```
// _volumeControl contains a pointer to the child
// VolumeControl object.

int VolumeDialog::getValue()
{
    return ( _volumeControl->getValue() );
}
```

Next, add the following function to the **VolumeControl** class:

```
int VolumeControl::getValue()
{
    return ( _scaleValue );
}
```

Finally, retrieve the value from the dialog with:

```
currentValue = _volumeDialog->getValue();
```

Using Containers

Once you have created a top-level window, you can "populate" it with interface elements. Because all of the top-level windows accept only one child element, that child element is almost always either a container or a complex component. This section describes how to choose appropriate containers to group and manage other elements. "Creating, Editing, and Manipulating Components" on page 89 discusses how to use components, but even in that case, you must understand how to choose an appropriate container to serve as the top-level element of a component. All of the containers are available on the Containers palette, shown in Figure 3-6, with the exception of the Tabbed Deck, which is on the ViewKit palette.

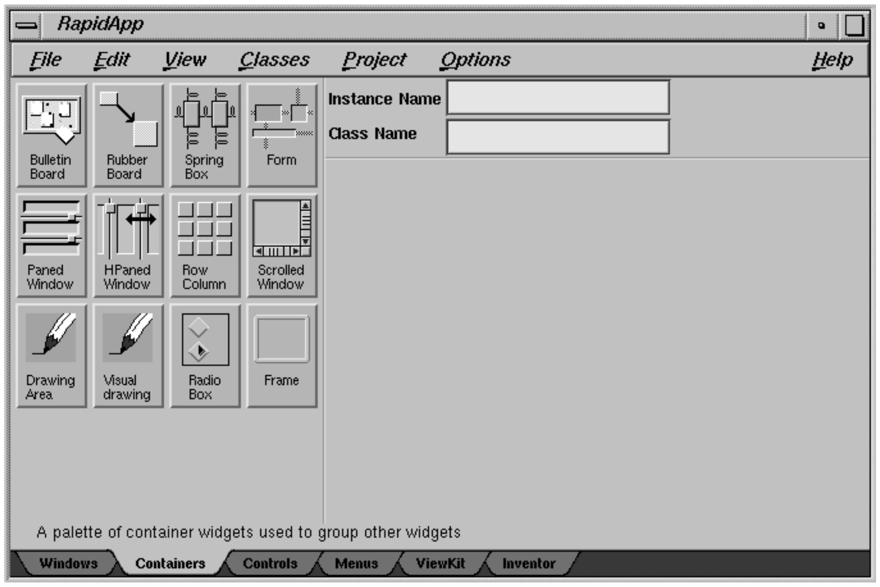


Figure 3-6 RapidApp Containers Palette

One of the challenges of working with IRIS IM is choosing an appropriate container to achieve the layout you would like. Many simpler systems give you only one type of container which requires you to place each component at a specific location within it. On these systems, if you want your interface to exhibit any type of dynamic behavior—allow users to resize windows, support internationalization (which requires dynamic layouts to handle different sized labels in different languages), allow users to customize portions of the user interface, and so on—you have to implement the support yourself.

IRIS IM does much more to help you with such requirements by providing a variety of containers that arrange their children in different ways. You can use IRIS IM containers to control the relationship of elements they contain. For example, you can left-align a group of elements, or you can create groups of elements such that the width of the largest element determines the width of the entire group. However, this flexibility adds more complexity. Instead of simply positioning widgets manually, you must position them by choosing and manipulating the right container. Furthermore, the IRIS IM containers were not designed with an interface builder in mind, and don't always behave as you might expect in response to interactive manipulation.

Note: Many containers add *constraints* to the elements they contain—resources that affect the appearance or behavior of an element within its container. These constraint resources appear on the children, not on the container. Different containers add different constraints, so you might see a constraint resource in one interface element and not in another of the same type if the elements are contained within different types of containers. The resource editor area lists constraint resources separately from other resources, below a label identifying them as constraint resources. You can modify constraint resources just as you do other resources.

Bulletin Board

The Bulletin Board widget (**XmBulletinBoard**) is the simplest IRIS IM container and the easiest to use. You simply "tack" elements to a particular position in the Bulletin Board and they stay there unless you explicitly move them. The Bulletin Board doesn't reposition or resize its children for any reason. Using a Bulletin Board container is the most like working with a drawing editor.

The limitation of a Bulletin Board is that all positions and sizes are fixed. For example, if you change the text or font of a label in a resource file, the label could grow or shrink, altering its alignment to other elements. Because of this limitation, the Bulletin Board is a poor choice for programs that you expect to internationalize or to allow users to customize the interface. Also, don't use a Bulletin Board if you want to allow the user to stretch or shrink the interface size. However, the Bulletin Board is a good choice for quickly prototyping interface designs because it is easy to use and provides the greatest flexibility for arranging elements within it.

Note: The Bulletin Board supports **marginWidth** and **marginHeight** resources that enforce minimum offsets from the edges of the container to its child elements. However, the Bulletin Board wasn't designed with an interactive builder in mind, so after initially placing an element you can move it closer to the edge of the Bulletin Board than allowed by the margin values. But when you run your application, the Bulletin Board overrides the children's positions and places them within the margins, resulting in a layout slightly different from what you specified in RapidApp. Therefore, either obey the margins when placing and moving elements, or change the **marginWidth** and **marginHeight** resources if you prefer smaller margins.

See the XmBulletinBoard(3Xm) reference page for more information on the **XmBulletinBoard** widget.

Rubber Board

The Rubber Board widget (**SgRubberBoard**) is an IRIS IM extension to Motif. The Rubber Board is similar to a Bulletin Board and shares both its ease of use and some of its limitations. However, it has a unique ability to support resizable layouts simply and easily. This widget is also designed explicitly for use with an interface builder; it would be awkward to use programmatically.

To use the Rubber Board:

1. Create an instance of it as small as you reasonably expect your window to be.
2. Place child elements on the Rubber Board just as you would a Bulletin Board.
3. Select the Rubber Board and toggle its **setInitial** resource to True.
4. Stretch the window until it is as large as possible (full screen is best).
5. Reposition and resize all the children so that the layout is as you would want it to appear if the user resized the window to that size.
6. Select the Rubber Board and toggle the **setFinal** resource to True. From this point on, the Rubber Board interpolates the positions and sizes of all its children as it resizes.

Note: The Rubber Board responds to changes in size initiated only by its parent (for example, its parent window); it doesn't respond to changes in the size of its children. Therefore, the Rubber Board continues to have the same limitations with respect to changing fonts, labels, or internationalization as the Bulletin Board.

The Rubber Board widget interpolates both size and position. In theory you can create bizarre dynamic behavior in which widgets move unexpectedly in response to resizing. For example, a widget on the right side of the Rubber Board when the container is small can move slowly to the left side as the Rubber Board grows larger. For obvious reasons, avoid using the Rubber Board in this manner.

You can nest Rubber Boards within one another. To do so, it's best to design the resize behavior of the inner containers first, and then place them in the larger Rubber Board.

Tip: The Rubber Board doesn't handle certain errors well, such as making the final size smaller than the initial size. Therefore, create the initial layout with the Rubber Board as small as possible, even if the size is unrealistic. Similarly, create the final size as large as possible.

Figure 3-7 through Figure 3-10 demonstrate the behavior of the Rubber Board widget. To begin, create an interface in a small container, such as in Figure 3-7. Once you've finished the layout, set the Rubber Board's **setInitial** resource to True.

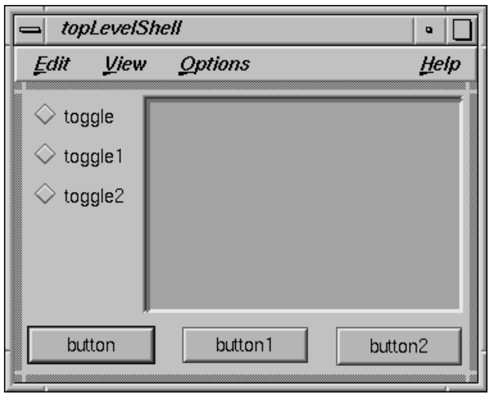


Figure 3-7 Rubber Board: Initial Layout

Next, resize the Rubber Board to a much larger size, as in Figure 3-8. Notice that all widgets keep their original size and position.

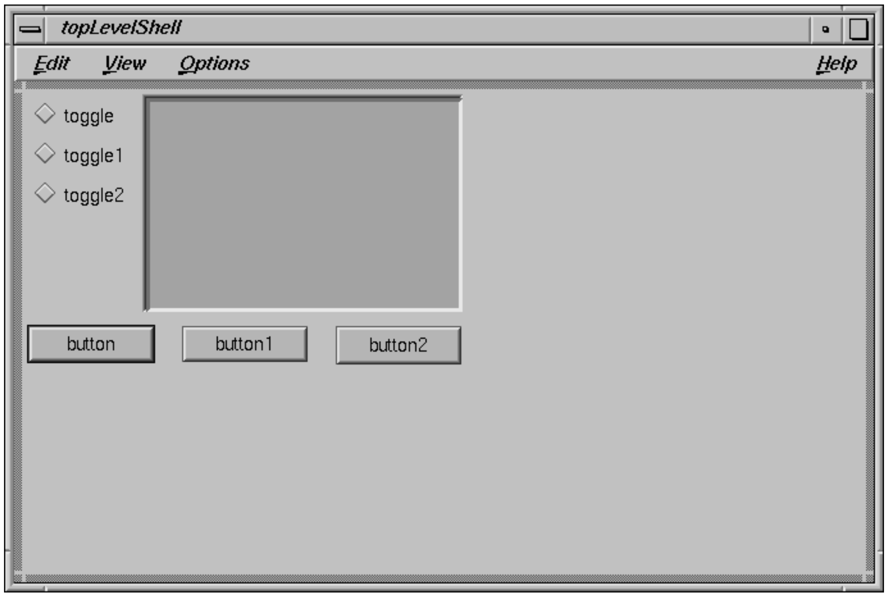


Figure 3-8 Rubber Board: Preparing for Larger Layout

Then resize and reposition the elements to reflect their desired size and position for the larger container size, as shown in Figure 3-9.

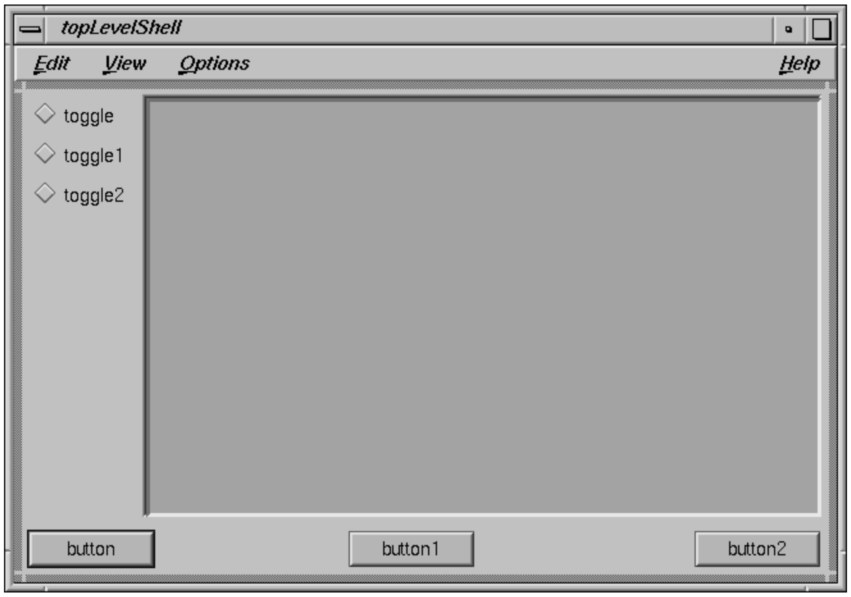


Figure 3-9 Rubber Board: Final Layout

After setting the Rubber Board's **setFinal** resource, you can resize the Rubber Board to any shape and the children will maintain their relative positions and sizes, as demonstrated in Figure 3-10.

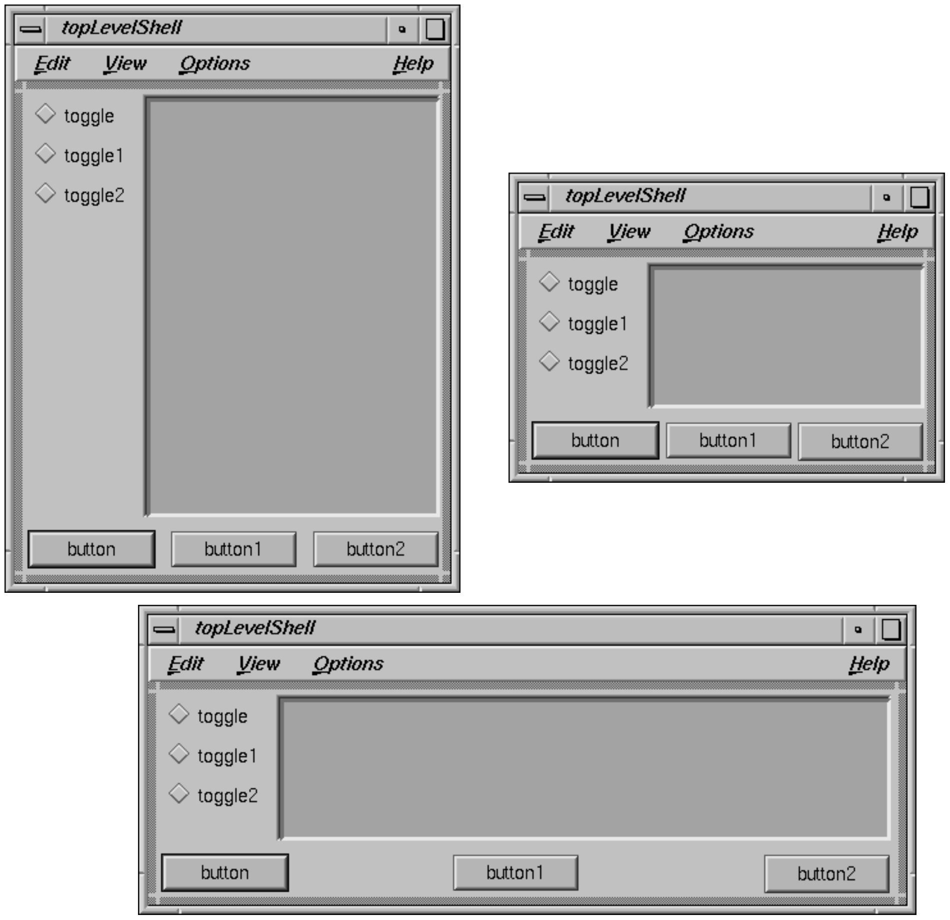


Figure 3-10 Effect of Resizing the Final Rubber Board Layout

Spring Box

The Spring Box widget (**SgSpringBox**) is an IRIS IM extension to Motif. At its simplest, the Spring Box simply enforces row or column behavior on its child elements. Figure 3-11 shows two simple layouts with buttons placed in vertical and horizontal Spring Boxes.

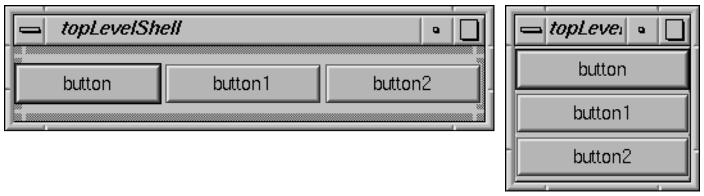


Figure 3-11 Vertical and Horizontal Spring Boxes

What you can’t see from Figure 3-11 is that each child of the Spring Box has six springs associated with it, as shown in Figure 3-12. Each spring has an associated “spring constant” value, which combines with the other springs to determine the overall behavior of the spring system. You can control each spring individually.

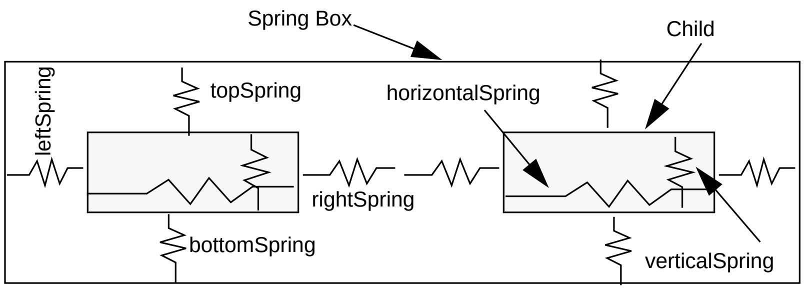
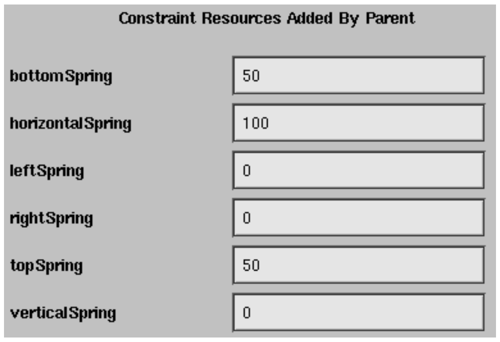


Figure 3-12 Springs in Children of a Spring Box

By default, the value of the horizontal and vertical spring resources are set to 100, while the other springs are set to 0. This means the children of the Spring Box stretch to fill the size of the Spring Box.

You can change the values of the spring resources by selecting a child and changing its constraint resources, as shown in Figure 3-13.


Constraint Resources Added By Parent

bottomSpring	50
horizontalSpring	100
leftSpring	0
rightSpring	0
topSpring	50
verticalSpring	0

Figure 3-13 Setting Spring Resources

For example, consider the behavior if you set up the following spring values:

	button1	button2	button3
leftSpring	0	0	0
rightSpring	0	0	0
topSpring	0	0	0
bottomSpring	0	0	0
verticalSpring	100	100	100
horizontalSpring	0	100	0

Figure 3-14 shows the layout created by these values, both when the Spring Box is its natural size and when it is stretched.

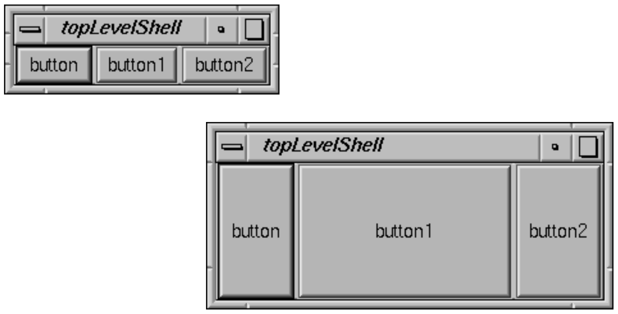


Figure 3-14 Spring Box Behavior With Modified Values

To create complete layouts using the Spring Box, you usually need to nest Spring Boxes within Spring Boxes, mixing vertical and horizontal orientations.

The Spring Box container uses the creation order of its children to determine their positions. You can move a child to a different position by selecting it and then using the “Up / Left” (or the **`<Ctrl+u>`**, **`<Left arrow>`**, or **`<Up arrow>`** keyboard shortcut) and “Down / Right” (or the **`<Ctrl+d>`**, **`<Right arrow>`**, or **`<Down arrow>`** keyboard shortcut) selections from the Edit menu.

The Spring Box tends to wrap itself tightly around its children, so that you can’t select or move it directly. To access the Spring Box, select a child of the Spring Box widget, then choose “Select Parent” from the Edit menu (or type the **`<Ctrl+p>`** keyboard shortcut) to select the Spring Box widget. You can then access the Spring Box’s resources. To move or resize the Spring Box, hold down the **`<Ctrl>`** key while using the left mouse button as you normally would. The **`<Ctrl>`** key prevents RapidApp from selecting a new element so that you can easily manipulate the currently selected element.

Form

The Form widget (**XmForm**) is the most common choice for resizable layouts. The Form widget positions its children based on attachments. For example, you can attach an element to a percentage position in the Form, to the side of another element, and so on. Forms can respond to resizes initiated by both its parent (for example, its parent window) and its children.

The traditional problem with Forms is that they are difficult to set up and use. Programmatically setting all the attachment resources for the Form’s children is tedious, and it’s difficult to envision the resulting appearance. RapidApp makes Forms easier to use by allowing you to interactively edit attachments and see the results.

Tip: Although you can create complex layouts within a Form widget, often it’s simpler to create simple layouts with only a few widgets, define that collection as a component, and then group the component with other components in a parent Form.

The easiest way to understand how to manipulate elements within a Form is by example. Figure 3-15 shows what happens when you add a push button to a Form.

Note the symbols around the push button and the lines from the button to the edge of the Form. The symbols are called *attachment icons*; there is one for each side of a child in a Form. The lines represent attachments. In this case, the button is attached to the top and left sides of the Form and is unattached on the right and bottom. This is the default behavior for an interface element placed in a Form.

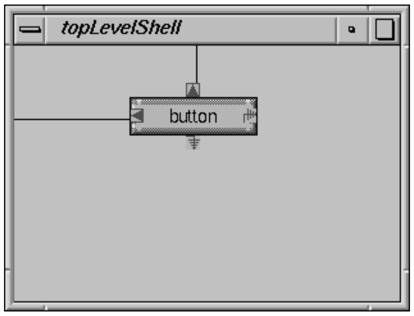


Figure 3-15 Push Button in a Form

The length of the line represents the offset from the point of attachment to the element. You can vary this offset in several ways. First, you can simply move the element. For example, moving the push button to the top of the window as shown in Figure 3-16 sets the top offset to zero.

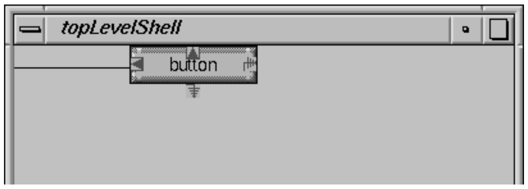
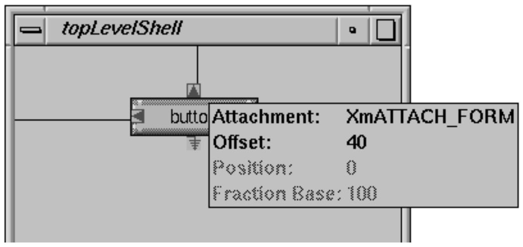


Figure 3-16 Setting the Top Offset to Zero

You can also set the offset by holding down the **`<Shift>`** key and pressing the left mouse button over an attachment icon. RapidApp displays a menu showing the value of the offset. You can change this value by moving the mouse while continuing to hold down the **<Shift>** key and left mouse button. Figure 3-17 shows an example.



Attachment: XmATTACH_FORM
Offset: 40
Position: 0
Fraction Base: 100

Figure 3-17 Using the Popup to Set an Offset

Alternatively, you can change the value of the offset in the appropriate field of the resource editor when the child element is selected.

You can change the type of an attachment by pressing the right mouse button over the attachment icon. RapidApp displays a menu showing the attachment type choices. For example, Figure 3-18 demonstrates pressing the right mouse button over the right attachment icon. Figure 3-19 shows the results of selecting XmATTACH_FORM.

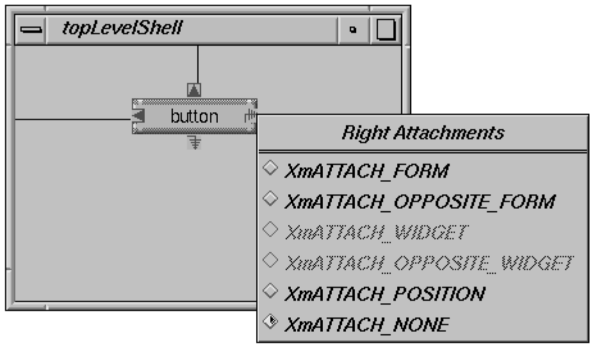


Figure 3-18 Displaying the Attachment Menu

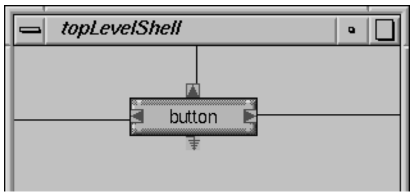


Figure 3-19 Push Button With a Right Attachment

Another way to create or edit an attachment is by dragging from an attachment icon to another interface element. For example, add a second push button to the Form, near the bottom of the container. Now press the left mouse button over the new button's top attachment icon and drag to the bottom edge of the original button, as shown in Figure 3-20.

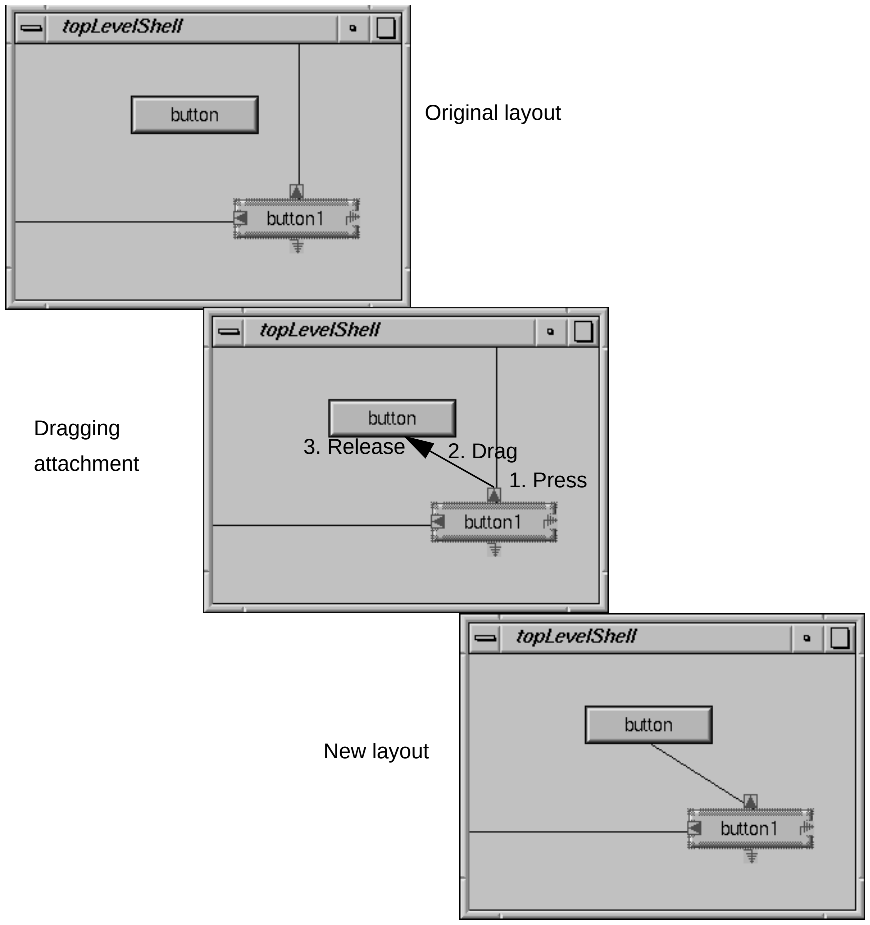


Figure 3-20 Drawing an Attachment

The XmATTACH_POSITION attachment type allows you to set the position of the element within a Form relative to the size of the Form. For example, you can specify the position of an element so that its top is always one quarter of the way from the top of the Form no matter what size the Form takes. To do this, you must specify two resource values: a numerator (in the interface element) and a denominator (in the Form). The denominator is the **fractionBase** resource in the Form. You can set the numerator either interactively, in the same way that you set the offset, or by changing the appropriate position resource when the child element is selected.

Note: If you use position attachments, be sure to set the value of the **fractionBase** resource before setting the attachments of any children to XmATTACH_POSITION. A Form doesn't recompute the children's position attachments if you change the Form's fraction base.

See the XmForm(3Xm) reference page for more information on the **XmForm** widget.

Paned Windows

The IRIS IM Paned Window widget (**XmPanedWindow**) places all its children in a column with each widget separated by a control, known as a sash, and an optional separator. The user can drag the sash to adjust the height of a section. The Paned Window widget adds constraint resources to each child that you can use to specify a minimum or maximum size. The Paned Window is suitable for interfaces that contain panels of information that the user might want to hide, reveal, or enlarge separately. See the XmPanedWindow(3Xm) reference page for more information on the **XmPanedWindow** widget.

The HPaned Window widget (**SgHorzPanedWindow**) is an IRIS IM extension to Motif. This widget is identical in functionality to the **XmPanedWindow** widget, but arranges its children in a horizontal row with separators and sashes between each child. Figure 3-21 shows an example of the HPaned Window.

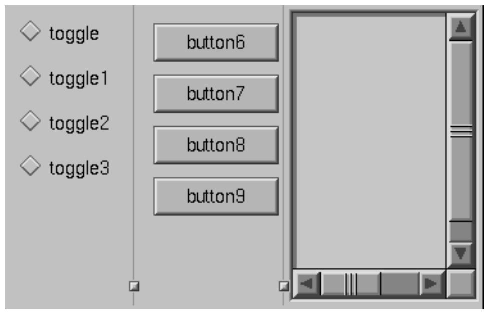


Figure 3-21 HPaned Window Container

The Paned Window containers use the creation order of their children to determine their positions. You can move a child to a different position by selecting it and then using the "Up / Left" (or the **`<Ctrl+u>`**, **`<Left arrow>`**, or **`<Up arrow>`** keyboard shortcut) and "Down / Right" (or the **`<Ctrl+d>`**, **`<Right arrow>`**, or **`<Down arrow>`** keyboard shortcut) selections from the Edit menu.

Because these Paned Window containers weren't designed with an interactive builder in mind, they might exhibit some odd behaviors in RapidApp.

- If you add a single child to the Paned Window, you can no longer click the Paned Window to edit its resources or add another child. To select the Paned Window, select a child of the Paned Window, then choose "Select Parent" from the Edit menu (or use the **`<Ctrl+p>`** keyboard shortcut). You can then access the Paned Window's resources. To move or resize the Paned Window, hold down the **`<Ctrl>`** key while using the left mouse button as you normally would. The **`<Ctrl>`** key prevents RapidApp from selecting a new element so that you can easily manipulate the currently selected one.
- The easiest way to add more child elements to the Paned Window is to select the Paned Window and to toggle on "Keep Parent" in the View menu. You can then add as many children to the Paned Window as you want. When you are finished adding children, toggle off "Keep Parent."
- After you add the first child, all subsequent children that you add have zero height. Furthermore, if you reorder the Paned Window's children, the Paned Window might resize some of the children, possibly even to a zero height. To get around this either: 1) as soon as you add a child, edit its **minHeight** resource to be a larger size; or 2) move the sash(es) so all children are the desired size.

RowColumn

The primary purpose of the RowColumn widget (**XmRowColumn**) is to support menu panes and menu bars. It also has limited use for simple aligned rows, and can support multiple columns as well. However, the RowColumn container forces all of its children to have the same height, and it provides only limited ability to control how children are resized. If you want a layout like that shown in Figure 3-22, then the RowColumn widget is a good choice. Otherwise, you might want to choose another container.

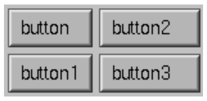


Figure 3-22 Typical RowColumn Layout

The RowColumn container uses the creation order of its children to determine their positions. You can move a child to a different position by selecting it and then using the "Up / Left" (or the **`<Ctrl+u>`**, **`<Left arrow>`**, or **`<Up arrow>`** keyboard shortcut) and "Down / Right" (or the **`<Ctrl+d>`**, **`<Right arrow>`**, or **`<Down arrow>`** keyboard shortcut) selections from the Edit menu.

The RowColumn container tends to wrap itself tightly around it children, so that it cannot be selected or moved. To select the RowColumn container, select a child of the RowColumn widget, then choose "Select Parent" from the Edit menu (or use the **`<Ctrl+p>`** keyboard shortcut). You can then access the RowColumn's resources. To move or resize the widget, hold down the **`<Ctrl>`** key while using the left mouse button as you normally would. The **`<Ctrl>`** key prevents RapidApp from selecting a new element so that you can easily manipulate the currently selected one.

See the XmRowColumn(3Xm) reference page for more information on the **XmRowColumn** widget.

Radio Box

The Radio Box container is really a RowColumn container. It enforces *radio behavior* (one-of-many) on all toggle buttons it contains. The Radio Box is useful for small rows or columns of one-of-many radio buttons, as shown in Figure 3-23.

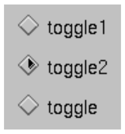


Figure 3-23 Radio Box With Toggle Button Children

When you create a Radio Box, RapidApp automatically creates two toggle buttons as children. In all other aspects, the Radio Box behaves the same as a RowColumn container (see “RowColumn” on page 79 for more information).

Frame

The Frame widget (**XmFrame**) is a purely decorative container, drawing a frame around its contents. A Frame can contain two children. One is the work area child, the widget surrounded by the Frame. The other is an optional label widget. The Frame places the label at the top, in-line with the frame, as shown in Figure 3-24. You can change its position slightly by editing the label’s constraint resources added by the Frame.

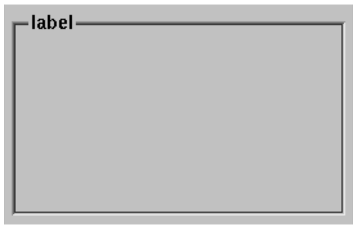


Figure 3-24 Frame Widget

Tip: It’s easiest to add the title label widget first. RapidApp initially places the label in the middle of the Frame as the work area child. You need to change the label widget’s **childType** constraint resource (added by the Frame) to XmFRAME_TITLE_CHILD. Once the title is in place, you can then add the work area widget for the Frame—typically, a container or a component.

See the XmFrame(3Xm) reference page for more information on the **XmFrame** widget.

Scrolled Window

The Scrolled Window widget (**XmScrolledWindow**) adds scroll bars to a child element. The Scrolled Window can contain only one child, typically a container or a component.

See the XmScrolledWindow(3Xm) reference page for more information on the **XmScrolledWindow** widget.

Drawing Areas

RapidApp provides two drawing area widgets: Drawing Area (**XmDrawingArea**) and Visual Drawing (**SgVisualDrawingArea**). These widgets provide a canvas on which you can draw using Xlib library calls. The Visual Drawing widget is an IRIS IM extension to Motif; it allows the widget to use a visual different from the rest of the application.

Although both drawing area widgets can function as simple containers, similar to the Bulletin Board, use these widgets only for drawing rather than managing other widgets. Other containers are more appropriate for managing child widgets.

See the XmDrawingArea(3Xm) reference page for more information on the **XmDrawingArea** widget. See the SgVisualDrawingArea(3Xm) reference page for more information on the **SgVisualDrawingArea** widget.

Tabbed Deck

The Tabbed Deck component is a special container available on the ViewKit palette. The Tabbed Deck arranges any number of child elements in a “deck.” The Tabbed Deck component displays only one child at a time, but also displays a tab area, with one tab for each child. The user can click a tab to display the corresponding child. Figure 3-25 shows an example of a Tabbed Deck.

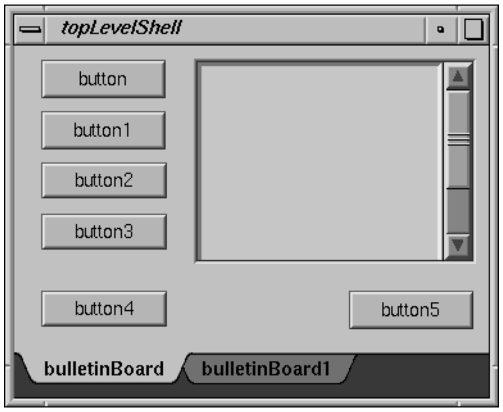


Figure 3-25 Tabbed Deck

You can add any number of child elements to the Tabbed Deck. Each element automatically fills the entire area of the Tabbed Deck except for the tab area.

Tip: After adding the first child element to a Tabbed Deck, add other elements by dropping them over the tab area.

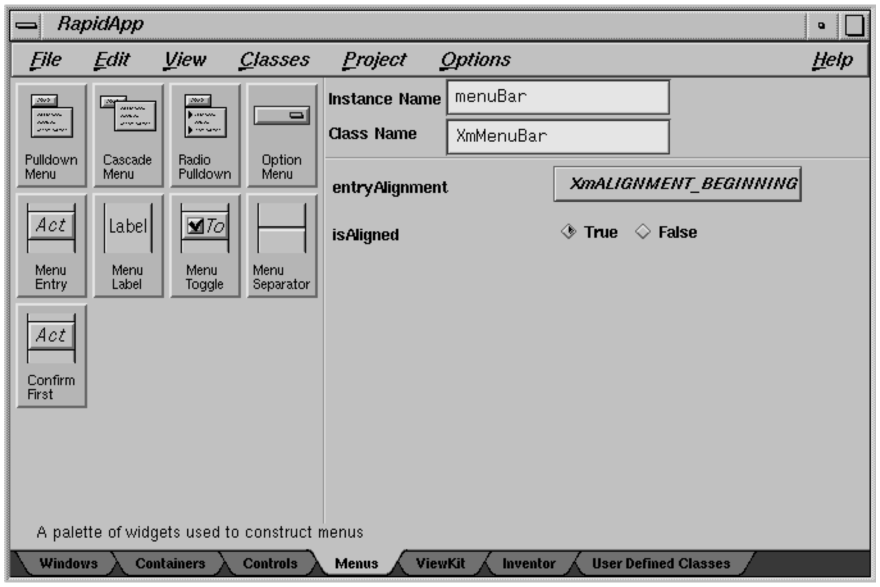
The Tabbed Deck creates a tab for each element you add. You can display an element, even in Build Mode, by selecting its corresponding tab. To change the text of an element's tab, select the element and edit the **tabLabel** constraint resource added by the Tabbed Deck.

When RapidApp generates code for a Tabbed Deck, it creates it as a subclass of the IRIS ViewKit **VkTabbedDeck** class. Furthermore, for each child of the Tabbed Deck that isn't a component (that is, a C++ class), RapidApp automatically encapsulates that child and its contents within a subclass of **VkComponent**. (See Chapter 2, "Components," in the *IRIS ViewKit Programmer's Guide* for more information on the VkComponent class.) The Tabbed Deck then simply creates an instance of that class.

Note: There is currently no way to reorder the children's positions within the Tabbed Deck. Be sure to add the children in the order in which you want them to appear in the tab area.

Creating and Editing Menus

The menus palette, shown in Figure 3-26, allows you to create menus and menu items. RapidApp allows you to create and manipulate both menu bars and option menus.



Instance Name: menuBar
Class Name: XmMenuBar
entryAlignment: XmALIGNMENT_BEGINNING
isAligned: [✓ True] False

A palette of widgets used to construct menus

Figure 3-26 RapidApp Menus Palette

Menu Bars

A menu bar consists of a collection of *cascade buttons* at the top of a window with pulldown menus (also referred to as menu panes) connected to them. This section describes how to create and edit menu bars using RapidApp. See “Menu Panes” on page 86 for information on editing the contents of individual menu panes.

Creating a Menu Bar

The only way to create a menu bar in RapidApp is to create a VkWindow. You can’t add a menu bar to a simple window after you create it.

Tip: If you build an interface in a simple window and later decide that you want a menu bar for the window, you can create a new VkWindow, cut or copy the top-level child (and thus everything it contains) of the existing simple window, and paste the interface into the new VkWindow.

When you create a VkWindow, RapidApp automatically includes a menu bar with many of the standard menu bar entries implemented complete with keyboard accelerators. See "VkWindows" on page 55 for more information on the standard menu bar entries.

Adding Panes to a Menu Bar

Add panes to a menu bar just as you add other elements to a container. First, select the menu bar. Then click with the left mouse button on the icon on the menus palette of the type of pane you want to add, then click with the left mouse button within the menu bar to add the item. Alternatively, you can click the icon with the middle mouse button, drag the item to the menu bar, then release the mouse button.

You can add the following two items to a menu bar:

Pulldown menu
: A regular menu pane. For your convenience, RapidApp automatically adds three initial menu entries to the pulldown menu. You can edit these items as described in "Menu Panes" on page 86.

Radio pulldown
: A menu pane that enforces *radio behavior* (one-of-many) on all toggles that it contains. For your convenience, RapidApp automatically adds three dummy menu toggles to the pulldown menu. You can edit these items as described in "Menu Panes" on page 86.

Tip: A convenient way to add multiple panes to a menu bar is to select the menu bar and then toggle on "Keep Parent" on the RapidApp View menu. RapidApp grays out all inapplicable items on the Menus palette, leaving active only those items you can add to a menu bar. You can then left-click an icon and drop it anywhere on the screen; RapidApp still adds the item to the selected menu bar.

After adding a menu pane, you can use the RapidApp resource editor to change the menu's label and mnemonic.

Removing Panes From a Menu Bar

Remove menu panes just as you do any other element in RapidApp. Simply select the cascade button in the menu bar for that menu pane, then cut it or delete it.

Moving Panes In a Menu Bar

To move a menu pane in a menu bar, select the cascade button in the menu bar for that menu pane, then use the "Up / Left" (or the **`<Ctrl+u>`**, **`<Left arrow>`**, or **`<Up arrow>`** keyboard shortcut) and "Down / Right" (or the **`<Ctrl+d>`**, **`<Right arrow>`**, or **`<Down arrow>`** keyboard shortcut) selections from the Edit menu.

Menu Panes

This section describes how to build individual menus—that is, the contents of individual menu panes.

Displaying and Hiding a Menu's Contents

When running an application, menus are transitory: they appear only when posted and disappear after the user makes a selection. Of course, this isn't useful when creating and editing menus, so RapidApp can display a menu continuously while you are constructing it.

Once you select a menu's cascade button, clicking on it again with the left mouse button causes RapidApp to display the menu's contents. Subsequent clicks toggle the display of the menu's contents off and on. Once you display the menu's contents, you can select and manipulate individual menu items as you would any other element in RapidApp. You can display multiple menus at once, and even drag and drop menu items between menus.

Adding Items to a Menu

You add items to a menu just as you add elements to a container (in fact, the menu pane container is simply a RowColumn widget). First, select the menu or any item in the menu. Then click with the left mouse button on the icon on the menus palette of the type of item you want to add, then click with the left mouse button within the menu to add the item. Alternatively, you can click the icon with the middle mouse button, drag the item to the menu, then release the mouse button.

You can add the following items to a menu:

Menu entry	A selectable action (implemented as a an XmPushButtonGadget)
Confirm first	A selectable entry that posts a confirmation dialog before executing the action. Confirm First menu items don't support an **undoCallback** resource.
Menu toggle	A selectable toggle entry. To enforce *radio behavior* on a group of toggles within a menu, put them within a Radio Pulldown menu.
Label	A non-selectable label.
Separator	A non-selectable separator.
Pulldown menu	A cascading, or pull-right, menu. For your convenience, RapidApp automatically adds three initial menu entries to the pulldown menu.
Radio pulldown	A cascading menu that enforces *radio behavior* (one-of-many) on all toggles that it contains. For your convenience, RapidApp automatically adds three initial menu toggles to the pulldown menu.

Tip: A convenient way to add multiple items to a menu is to select the menu (select any item in the menu, then choose "Select Parent" from the RapidApp Edit menu), then toggle on "Keep Parent" on the RapidApp View menu. RapidApp grays out all inapplicable items on the Menus palette, leaving active only those items you can add to a menu bar. You can then left-click an icon and drop it anywhere on the screen; RapidApp still adds the item to the selected menu.

After adding an item, you can use the RapidApp resource editor to change the item's label and mnemonic. For each item that invokes an action—Menu Entry, Confirm First, and Menu Toggle—you must define an **activateCallback** function that your application invokes when the user selects the item. For Menu Entry and Menu Toggle items, you can also define an **undoCallback** function that your application can invoke to undo the effects of the item's action.

For each menu item in a menu pane, RapidApp adds a member function of the same name to the VkWindow's child component. You can then add the functional code to the functions to implement the menu items.

Moving Items in a Menu

To move an item in a menu, select the item and use the "Up / Left" (or the **`<Ctrl+u>`**, **`<Left arrow>`**, or **`<Up arrow>`** keyboard shortcut) and "Down / Right" (or the **`<Ctrl+d>`**, **`<Right arrow>`**, or **`<Down arrow>`** keyboard shortcut) selections from the Edit menu.

Removing Items From a Menu

Remove items from a menu just as you do other elements in RapidApp. Simply select the menu item, then cut it or delete it.

Option Menus

An option menu is an interface element that allows the user to select one of several options using a menu. An option menu consists of a label and the equivalent of a cascading menu. When not displaying the cascading menu, an option menu displays the last item the user selected.

You create an option menu just as you do other interface elements. A newly created option menu has no label. To work with an option menu more easily, immediately edit the option menu's **labelString** resource to provide a label.

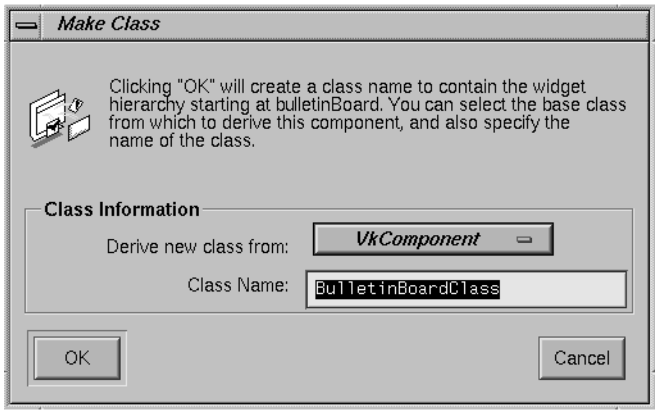
You can click anywhere on the option menu's label or cascade button to display its cascading menu pane. RapidApp automatically adds two dummy menu entries to the option menu when you create it. You can edit the option menu pane as described in "Menu Panes" on page 86.

Creating, Editing, and Manipulating Components

An important concept in RapidApp is creating self-contained components that you can then reuse not only in the application you're currently building, but in other applications as well. This section describes how to create and edit components in RapidApp.

Creating Components

It's easy to create components in RapidApp. Simply select the container that you want to be the top-level element in your component, and choose "Make Class" from the Classes menu. RapidApp displays a dialog prompting you for the name of your new class, as shown in Figure 3-27. After creating the new class, RapidApp adds an icon for it on the User Defined Classes palette. (RapidApp creates the palette if it doesn't already exist.)


Make Class

Clicking "OK" will create a class name to contain the widget hierarchy starting at bulletinBoard. You can select the base class from which to derive this component, and also specify the name of the class.

Class Information

Derive new class from: VkComponent

Class Name: BulletinBoardClass

OK

Cancel

Figure 3-27 Make Class Dialog

By default, RapidApp creates your component as a subclass of the IRIS ViewKit **VkComponent** class. If you want to handle ToolTalk messages with your component, use the option menu on the dialog to tell RapidApp to derive the new class from the IRIS ViewKit **VkMsgComponent** class.

Note: If you want to handle ToolTalk messages in your application, you also need to select "Application" from the Options menu and toggle on the "Use ToolTalk" option. This causes RapidApp instantiate a **VkMsgApp** object instead of a **VkApp** object. See Appendix A, "ViewKit Interprocess Message Facility," in the *IRIS ViewKit Programmer's Guide* for more information on the **VkMsgApp** class and the IRIS ViewKit support for ToolTalk.

As discussed in "Object-Oriented Design" on page 42, RapidApp generates two separate C++ classes. One class, which usually has the suffix UI appended to the class name, contains all the code needed to generate the user interface, including creating components and IRIS IM widgets, registering callbacks, and so on. The second generated class is a subclass of the UI class and contains the code that implements the actual functionality of the component. When you add the functional code to your component, you should do so in only the derived class.

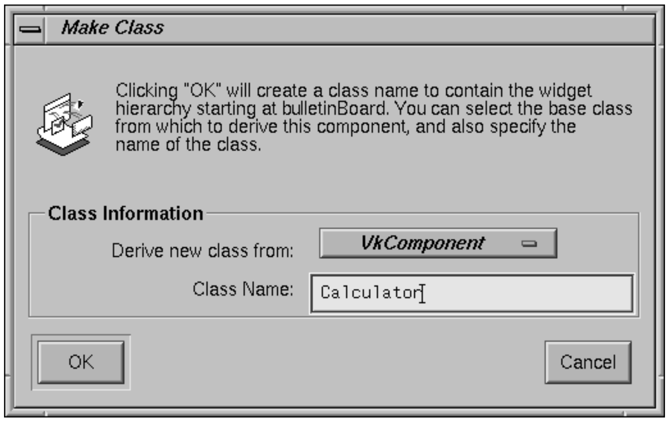
As an example of creating a component, consider the calculator program created in "Example: A Calculator" on page 24. You can encapsulate the calculator interface in a self-contained **Calculator** component so that you can reuse it elsewhere. To do so:

1. Open the file *calc.uil* in RapidApp.

   If RapidApp is running, select "Open" from the File menu, then select *calc.uil* from the file dialog that appears. If RapidApp isn't running, you can drag the *calc.uil* file onto the RapidApp icon, double click on the *calc.uil* icon, or change into the *Calc* directory and enter:

   ```
   % rapidapp calc.uil
   ```

2. Select the Bulletin Board container by clicking the background area of the calculator window.
3. Create a **Calculator** class by selecting "Make Class" from the Classes menu. Type "Calculator" in the prompt window that appears, as shown in Figure 3-28.



Clicking "OK" will create a class name to contain the widget hierarchy starting at bulletinBoard. You can select the base class from which to derive this component, and also specify the name of the class.

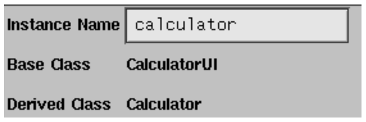
Class Information

Derive new class from: VkComponent

Class Name: Calculator

Figure 3-28 Creating a Calculator Class

RapidApp updates the resource editor area and header area to display information about the **Calculator** class that you created. Notice that the header area displays two names: **Calculator**, the name you provided; and **CalculatorUI**, as shown in Figure 3-29.



Instance Name	calculator
Base Class	CalculatorUI
Derived Class	Calculator

Figure 3-29 Class Header

RapidApp also creates a new palette named “User Defined Classes” (if it didn’t already exist). If you select that palette, you’ll notice it contains a new icon named “Calculator,” as shown in Figure 3-30. You can now create additional instances of the **Calculator** component just as you would any other interface element.

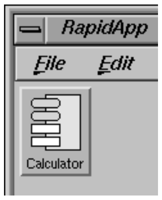


Figure 3-30 User Defined Classes Palette

4. Generate Code.

   If you generate code now, RapidApp creates a **CalculatorUI** and **Calculator** class. It no longer generates a **BulletinBoard** class as it did before because the top-level element in the window is already a class—**Calculator**. Because **Calculator** is new class, RapidApp can't merge the code changes you had previously made to implement the calculator functionality; that code is in the *BulletinBoard.C* file. You need to copy your changes from *BulletinBoard.C* to *Calculator.C*. RapidApp automatically updates the rest of your application to use the **Calculator** class rather than the **BulletinBoardClass** class.

Using Components

After creating a user-defined component, you can create instances of, select, move, and otherwise manipulate it just as you would any other interface element.

Editing Components

After creating a component, you can no longer simply click on one of its elements to edit it; RapidApp treats the component as a single interface element. However, you can still use RapidApp to edit the component.

To do so, toggle on "Edit Classes" in the Classes menu. RapidApp hides your current interface and displays all classes currently on the User Defined Classes palette. You can now select, edit, and manipulate the individual elements composing the classes. You can even add elements to and delete them from components. When you are finished editing classes, toggle off "Edit Classes." RapidApp redisplays your current interface, reflecting the changes you made to your components.

Deleting Components

Once you create a class, it remains on the User Defined Classes palette even if there are no instances of the class in your interface. When you save your interface, RapidApp saves the class along with the rest of the information about the interface. If you no longer want to keep a class on the palette, you can delete it in one of two ways:

- The first method is to "unmake" the class. To do so, create an instance of the class, select it, and then select "Unmake Class" from the Classes menu. RapidApp displays a dialog asking you if you want to remove the class from the palette. If you press *OK*, RapidApp removes the class; otherwise it "dismantles" the instance you have currently selected but leaves the class on the palette.
- The second method is to toggle on "Edit Classes" in the Classes menu. RapidApp hides your current interface and displays all classes currently on the User Defined Classes palette. Delete any class you no longer want by deleting the top-level window for that class. RapidApp then removes the class from the palette. Toggle off "Edit Classes" when you're finished deleting classes.

Creating Components from External Classes

Sometimes you might want to use interface classes that you didn't create with RapidApp. For example, you might have created components directly with IRIS ViewKit and would like to incorporate them in your application.

There is a simple method for using external classes in RapidApp. To illustrate this technique, assume that you wanted to use the **VkPie** class, but **VkPie** wasn't provided on the ViewKit palette. To include this class:

1. Create a container of any type.
2. Select the container, then select "Make Class" from the Classes menu.

3. In the Make Class dialog, enter the name of the external class you want to use. In this example, enter “VkPie.”
4. Position the component as you would like the external class to appear in your interface.
5. Toggle on “Edit Classes” in the Classes menu.
6. Select your “fake” component.
7. In the header area, toggle off the *Generate Code* toggle.

   With *Generate Code* toggled off, RapidApp won’t generate code for the component. Instead, you can include the header for your external class and link with the appropriate object file or library.

You can use this technique with any external class as long as the classes constructor follows the same calling conventions as the components created by RapidApp. As for all other subclasses of the IRIS ViewKit **VkComponent** class, the constructor should take two arguments. The first one is a character string, which should be used as the component’s name. The second is a Widget, which should be used as the component’s parent. See Chapter 2, “Components,” in the *IRIS ViewKit Programmer’s Guide* for more information on the **VkComponent** class.

Note: Components “imported” in this manner can’t be used as direct children of a top-level window in RapidApp.

Chapter 4

Example Programs

This chapter shows you some example applications created with RapidApp.

Chapter 4

Example Programs

In addition to IRIS IM widgets and IRIS ViewKit components, RapidApp also supports components from various Silicon Graphics libraries such as Open Inventor. This chapter shows examples of building applications with RapidApp using these libraries.

A Simple Open Inventor Program

This section shows how to use an Open Inventor component in an application. It demonstrates using both a text editor and the Debugger's Fix and Continue feature to add functionality to the code. In practice, you can use whichever method you prefer.

Note: Open Inventor is an optional product. You can't build this example if you don't have the *inventor_dev* package installed on your system.

The following example creates a simple interface with the Examiner Viewer to display a scene:

1. Create a Simple Window.
2. Create a Bulletin Board within the Simple Window.
3. Create an Examiner Viewer within the Bulletin Board. Make sure that the Instance Name of the component is "viewer."

   At this point, your window should look like the one shown in Figure 4-1.

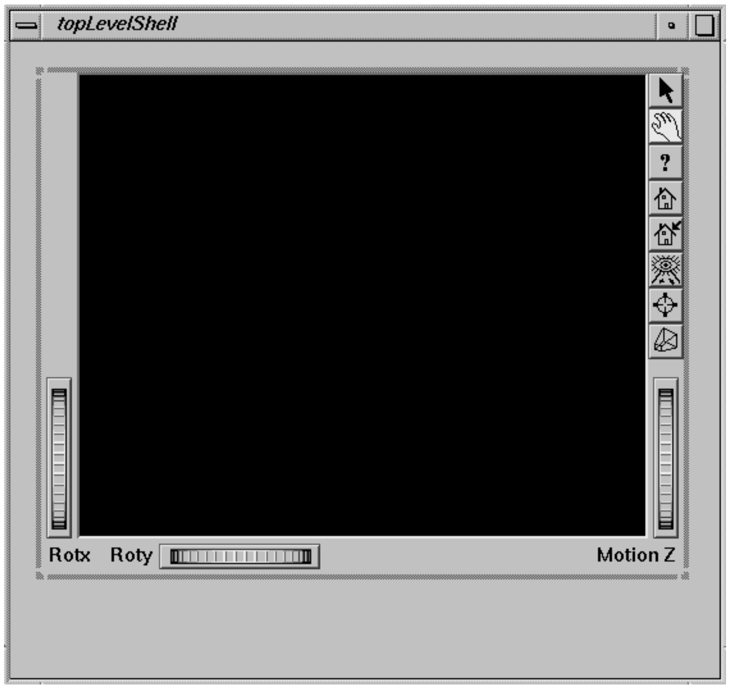


Figure 4-1 Adding an Examiner Viewer

4. Complete the interface by adding two Toggle Buttons below the viewer.
   - Create a Toggle Button and place it below and at the left side of the viewer. Change the label of the toggle to read “Headlights On.” Set the **set** resource to True so that the toggle is on by default.
   - Create another Toggle Button and place it to the right of the first toggle. Change the label of the toggle to read “Show Decorations.” Set the **set** resource to True so that the toggle is on by default.
5. Select the Bulletin Board and choose “Make Class” from the Classes menu. Name the class “ConePanel.” Figure 4-2 shows the completed interface.

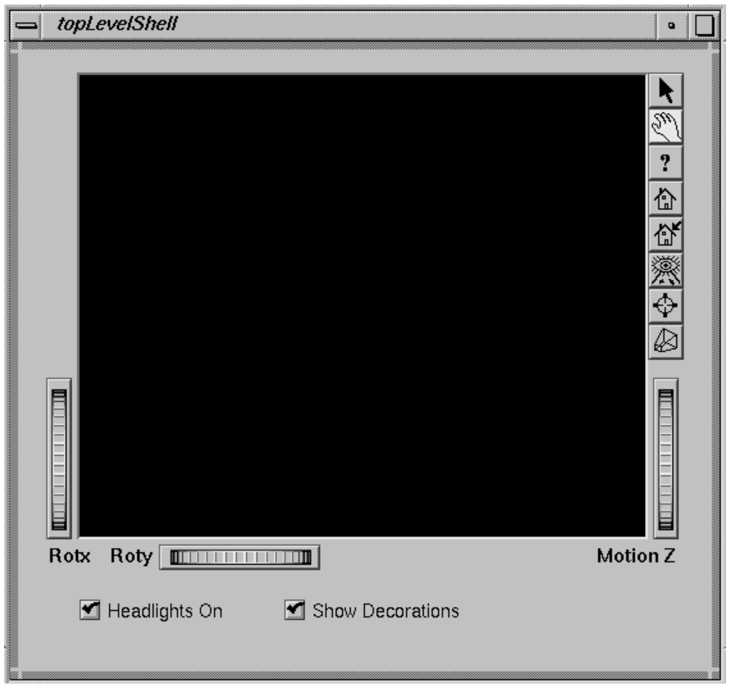


Figure 4-2 The Completed Open Inventor Component

6. Select “Application” from the Options menu. In the Application Names dialog that appears, change the application name to “cone” and the application class to “Cone.”
7. Go to the Desktop or a shell window and create a directory. Select “Save As” from the File menu and save the interface to *cone.uil* in the directory you created.

   Note: If you provide a directory name in the Application Names dialog, RapidApp creates the directory for you automatically if it doesn’t already exist.
8. Select “Generate C++” from the Project menu to generate code. Then selection “Build Application” from the Project menu to build the application.
9. Edit the ConePanel component to display the cone:

- Select “Edit File” from the Project menu and in the select the file *ConePanel.C*.
- Scroll down until you see a code segment that looks like this:

```
//---- ConePanel Constructor

ConePanel::ConePanel(const char *name, Widget parent) :
                   ConePanelUI(name, parent)
{
    // This constructor is called after the component’s interface has been built.

    //--- Add application code here:


} // End Constructor
```

- Go to the line after the “Add application code here” comment and type:

  `_viewer->setSceneGraph(new SoCone);`

- Save the file and exit the editor.
- Select “Build Application” from the Project menu. When the compilation has finished, select “Run Application” from the project menu. The window should look like Figure 4-3.

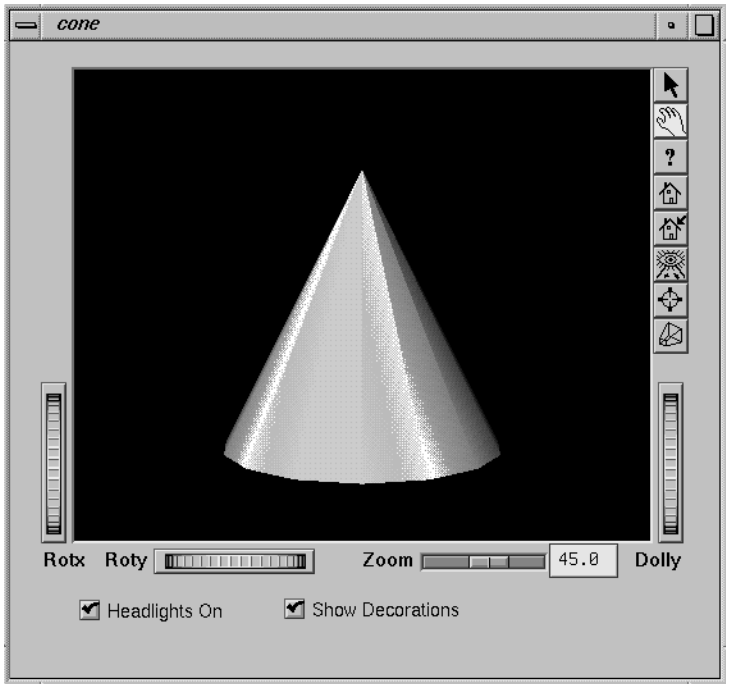


Figure 4-3 The Open Inventor Interface Displaying a Scene

10. Associate actions with the toggle buttons. Because you didn't assign any callbacks to the toggle buttons when you created the interface, you need to go back and add them. To do so:
    - Toggle on "Edit Classes" from the Classes menu.
    - Select the headlights toggle button and change its **valueChangedCallback** resource to "headlight()."
    - Select the decorations toggle and change its **valueChangedCallback** resource to "toggleDecorations()."
    - Toggle off "Edit Classes" from the Classes menu.
    - Select "Save" from the File menu to save the interface, and select "Generate C++" from the Project menu to generate code.
    - Select "Debug Application" from the Project menu to start the Debugger.

- Click the Debugger’s *Run* button to start your program.
- Click on the *Headlights On* toggle button. The Debugger stops in the **VkUnimplemented()** function.
- Click the Debugger’s *Return* button to go up one level to the callback function that invoked **VkUnimplemented()**. You’ll see a function that looks like this:

```
void ConePanel::headlight ( Widget w, XtPointer callData )
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct*) callData;

    //--- Comment out the following line when ConePanel::headlight is implemented:

    ::VkUnimplemented ( w, "ConePanel::headlight" );


    //--- Add application code for ConePanel::headlight here:


}
```

- Select “Edit” from the Debugger’s Fix+Continue menu and edit the function to look like this:

```
void ConePanel::headlight ( Widget w, XtPointer callData )
{
    XmToggleButtonCallbackStruct *cbs = (XmToggleButtonCallbackStruct*) callData;

    //--- Comment out the following line when ConePanel::headlight is implemented:

    //::VkUnimplemented ( w, "ConePanel::headlight" );


    //--- Add application code for ConePanel::headlight here:

    _viewer->setHeadlight(cbs->set);

}
```

Note: Remember to comment out the **VkUnimplemented()** call.

- Select “Parse And Load” from the Debugger’s Fix+Continue menu.
- Click the Debugger’s *Continue* button to continue the program. Try out the changes by clicking the program’s *Headlights On* button.

- Click on the *Show Decorations* toggle button. The Debugger stops in the **VkUnimplemented()** function.
- Click the Debugger's *Return* button to go up one level to the callback function that invoked **VkUnimplemented()**. You'll see a function that looks like this:

```
void ConePanel::toggleDecorations ( Widget w, XtPointer callData )
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct*) callData;

    //--- Comment out the following line when ConePanel::toggleDecorations is implemented:

    ::VkUnimplemented ( w, "ConePanel::toggleDecorations" );


    //--- Add application code for ConePanel::toggleDecorations here:


}
```

- Select "Edit" from the Debugger's Fix+Continue menu and edit the function to look like this:

```
void ConePanel::toggleDecorations ( Widget w, XtPointer callData )
{
    XmToggleButtonCallbackStruct *cbs = (XmToggleButtonCallbackStruct*) callData;

    //--- Comment out the following line when ConePanel::toggleDecorations is implemented:

    //::VkUnimplemented ( w, "ConePanel::toggleDecorations" );


    //--- Add application code for ConePanel::toggleDecorations here:

    _viewer->setDecoration(cbs->set);

}
```

- Click the Debugger's *Continue* button to continue the program. Try out the changes by clicking the program's *Show Decorations* button.
- Save the changes and rebuild the application.

Online Examples

If you install the *RapidApp.sw.examples* subsystem, the directory */usr/share/src/RapidApp* contains several example programs created using RapidApp. You can build the programs by entering "make" in the desired directory. You can also load any example in RapidApp. If you use the Indigo Magic Desktop, you can open a window in the example directory and simply double click on the file with the ".uil" suffix (the file with the RapidApp icon).

To run the programs, either run then from RapidApp, or be sure to set the XUSERFILESEARCHPATH environment variable to include "%N%S" so that you pick up the application resource files from the current directory.

To identify which code was created using RapidApp and which was added by hand, choose "View Changes..." from the Project menu and select a file. RapidApp displays a windows showing the differences (if any) between the current file and what RapidApp originally created. In general, the only files that have changes are subclasses, which you can identify by the existence of a similar file with a "UI" appended to the name. For example, *FooUI.C* is a base class generated by RapidApp, while *Foo.C* is a subclass that is likely to be modified from the original.

You might also want to explore the source code using the Developer Magic Static Analyzer and Class Browser. If so, be sure you have installed the *RapidApp.sw.examples_sadb* subsystem. Then start RapidApp, load the desired example, and toggle on "Create Static Analysis Database" in the Application Preference dialog. Then select "Browse Source" from the Project menu to launch the Static Analyzer on the example program.

The examples in */usr/share/src/RapidApp* include:

Calculator	A very simple calculator that adds two numbers and reports the result. This example was taken from a Visual C++ manual. There are two versions of this program, one that demonstrates the VkEZ interface and another that shows the equivalent IRIS IM version. You may find it interesting to view the differences using *xdiff*, as follows:

```
% cd Calculator; xdiff EZ/Calculator.C Motif/Calculator.C
```

DialCalc	A program that is similar to Calculator, but that uses two Dial widgets to enter numbers. A third dial shows the sum of the first two. This example creates each labeled dial element as a class, and shows how you can nest and connect components. In this case the program uses the IRIS ViewKit callback mechanism. Like Calculator, there are two versions of this program, one that uses the VkEZ interface, while the other uses the IRIS IM interface.
HelloCone	An example of using RapidApp and IRIS ViewKit with Open Inventor. This example is based on the HelloCone example in *The Inventor Mentor*.
Rectangle	An example of using an IRIS IM DrawingArea widget. The program creates a class that handles its own rendering, in this case simply drawing a large rectangle in the window.
OpenGL	Various examples using OpenGL and IRIS ViewKit.
Simple	This program is taken from *4DGifts*, where it demonstrates the OpenGL widget. This version uses IRIS ViewKit, and was created with RapidApp, but the OpenGL rendering code is mostly unchanged from the original.
IvViewer	Another IRIS ViewKit / Open Inventor example. This program allows you to open an Open Inventor data file and view it.
Stopwatch	A simple stopwatch program. This example is based on the stopwatch example in *Object-Oriented Programming with C++ and OSF/Motif*, but rewritten for IRIS ViewKit using RapidApp. The program provides an example of using multiple components and defining connections between them.
Convert	A program that converts between various number formats.

Appendix A

Frequently Asked Questions about RapidApp

This appendix provides tips for using RapidApp.

Frequently Asked Questions

The following are frequently asked questions (FAQs) and answers about RapidApp operation:

- Why don't any of my labels appear when I run programs created using RapidApp?

  X uses a resource file, commonly called an "app-defaults" file, to store various "resources" that affect widgets. It is good practice to put labels and other similar resources in a resource file rather than hard-coding them in your program, and RapidApp does this for you automatically. Unfortunately, X applications look for these files in many places, but not in the current directory. There are three ways to fix this:

  1. Run the application from RapidApp's project menu when you want to test it. RapidApp makes sure the resources are found.
  2. Set the environment variable XUSERFILESEARCHPATH to "%N%S", to add the current directory to the application's search path. You might want to do this as part of your login setup.
  3. Install the app-defaults file in /usr/lib/X11/app-defaults/. If you do a "make install", the Makefile generated by RapidApp will do this for you automatically. If you choose this approach, remember to reinstall any time you make changes.

  Approaches 1 and 2 are recommended.

- Why can’t I resize an option menu using RapidApp?

  You can, you just don’t see what you expect. The IRIS IM Option menu is really a row column widget with a menu floating inside it. You can resize the outer row column widget, but the inner, visible option menu is totally controlled by the row column. It is not accessible to RapidApp and cannot be resize.

- Why can’t I resize a scale widget using RapidApp?

  Like option menus, you can, you just don’t see what you expect. The IRIS IM Scale widget is really a container with a Scrollbar floating inside. You can resize the outer portion of the Scale widget, but the inner, visible Scrollbar is totally controlled by the Scale. It is not accessible to RapidApp and cannot be resize. The size of the Scrollbar is controlled by the **scaleWidth** and **scaleHeight** resources.

- How do I add my own files to the makefile generated by RapidApp?

  The Makefile builds all files listed in C++FILES. This variable is defined as the contents of two lists, BUILDERFILES and USERFILES. For best results with the merging feature of the code generation, you should add externally-created files to the USERFILES list. Just list the source files, the Makefile will do the rest.

- How do I create icons for bitmaps or pixmaps?

  You can use any number of external tools. For example, bitmaps can be created with the “bitmap” program, which is distributed with *x_eoe*. Color icons can be created any drawing editor, and converted if necessary using the *ppm* or similar utilities. There is a program called *xpaint* available as part of the original IndiZone package, which is very nice for editing pixmaps. The public domain *xv* editor is also useful for reading and processing various file types and writing the Xpm format recognized by IRIS IM (and RapidApp).

- How do I add a menu bar to my window?

  If you want a window to have a menu bar, start with the VkWindow element found on the windows palette. You can add, remove, or alter the built-in menus as you wish.

- How do I add items to a menu?

  First, you have to access the menu pane itself. To do this, select the entry on the menu bar, or the option menu, depending the type of menu you are working with. Then click again. The menu pane will be displayed. Now you can drop new elements onto the pane. Note that the hotspot when dropping items is the mouse pointer. Dismiss the menu by clicking again on the element that launched the menu.

- Why do container widgets sometimes change size when I add or remove children?

  IRIS IM uses an algorithmic approach to layout. Each container has its own algorithm for arranging its children, which is triggered each time a child is added or removed. So even though you may have specified a size for a container, the container itself may recompute and change this size when the contents change. This is central to the behavior of IRIS IM and there is little that can be done to improve this behavior.

- Why can’t I add a button to a VkWindow object?

  IRIS IM offers a wide variety of layout styles, which can be selected by choosing an appropriate container or set of containers. The top-level window elements in RapidApp do not enforce any one container, so you can choose your own. Each of the elements on the Windows palette may contain exactly one child, which must be a container of your choice.

  If you are expecting the very basic behavior of some PC layout programs, add a Bulletin Board container to the top-level window. For more advanced needs, you might add a Form, a Paned Window, or other container. For a graphics oriented program, you might add a drawing area, GL widget, or Inventor scene viewer. Once you have selected your basic container, you can add child elements to it.

- How do I create a standard IRIS IM dialog?

  Dialogs tend to be dynamic by nature. Although RapidApp could allow you to associate and information dialog (for example) with a button in such a way that the dialog is posted when the button is pressed, this is rarely useful in real programs. Although there are exceptions, dialogs are generally posted in response to some condition that can only be determined at run-time. Therefore, RapidApp doesn’t bother to provide these dialogs.

However, because RapidApp generates ViewKit programs, it is easy to add these dialogs programmatically along with the logic associated with the condition that requires a dialog. For example, to post a warning dialog, simple write:

```
theWarningDialog->post("Warning, serious problem detected");
```

To ask a question that requires an answer, write:

```
if (theQuestionDialog->postAndWait("Really exit?") == VkDialogManager::OK)
    exit(0);
```

For more information, see the *IRIS ViewKit Programmer's Guide*.

- How do I create a custom dialog?

  You can create your own dialogs by selecting a ***VkDialog*** as the top-level window for an interface. Add a single container and design the contents of your dialog. The resulting class will be derived from ***VkDialogManager***, and support the same API as other ViewKit dialogs.

- How do I add a button to the dialog class?

  In RapidApp, you cannot. The ***VkDialogManager*** class and all subclasses allows you to determine what buttons appear by how the dialog is posted. See the ViewKit Programmer's Guide for more information.

- How can I launch my own editor from RapidApp?

  If you set the environment variable $WINEDITOR, and select the "Use $WINEDITOR" option from the Preferences panel, your editor will be launched from RapidApp's "Edit Files..." menu.

- What is VkEZ and why would I use it?

  VkEZ is a very simple set of wrappers that can be attached to a widget at run-time to provide an easy-to-remember API for manipulating widgets. VkEZ offers many of the benefits of "widget wrappers", in that they offer a C++-like API for manipulating widgets, without committing yourself to basing your entire program to yet another layer above IRIS IM. VkEZ is intended for quick prototyping, and like any wrapper approach has a cost over and above the normal API. Therefore, it should be replaced in any code that demands efficiency.

VkEZ allows you to substitute easy-to-remember code segments for more complex widget code. For example, assume you want to extract an integer value from a text field widget, add it to the current value of a scale widget, and display a running trace of these values in a scrolled text widget. Using the IRIS IM API, you could write something like:

```
/* IRIS IM API version */
int value1, value2;
char buf[100];
value1 = atoi(XmTextGetString(_textfield));
XtVaGetValues(_scale, XmNvalue, &value2, NULL);
sprintf(buf, “%d”, value1 + value2);
XmTextInsertString(_scrolledtext,
XmTextGetInsertionPosition(_scrolledtext), buf);
```

The EZ equivalent would be:

```
EZ(_scrolledtext) << (int)EZ(_textfield) + EZ(scale);
```

- If I edit the code produced by RapidApp and then need to change the interface, will I lose my hand-edited changes?

  RapidApp merges all code that is generated with existing code using a 3-way merge tool that should automatically merge reasonable changes automatically. In addition, RapidApp always makes a backup of your original file when files are merged. If the merge is unsuccessful, RapidApp displays *xdiff*, a tool that allows you to resolve any problem areas by hand. You can help the merge process proceed smoothly by not making gratuitous changes (like reformatting for style) to the generate code, placing large bodies of code in external files, and so on.

- Sometimes I drop a widget on a location, and I get a new window instead of the widget going where I want it to go. Why?

  Some widgets cannot be children of other widgets. If you drop a button on a button, for example, the dropped button cannot be created as a child of the drop site. When this happens, RapidApp searches for a valid container up the widget hierarchy. If none is found, the element is created as a top-level window.

- How do I move menu elements within a menu pane?

  You can use the Up/Left or Down/Right commands in the Edit menu, or use the arrow keys to reorder elements in a menu, or any row column widget. You can also drag and drop items between menus using the middle mouse button.

- How do I move window elements between menu panes?

  You can use Cut and Paste, or you can drag and drop items between menus using the middle mouse button. Note the middle mouse drag, while holding down the Control key performs a copy.

- I want to apply a resource to multiple widgets. Is there an easy way to do that?

  There is no way to select multiple widgets and no way to apply a resource to a group of existing widgets. However, if you plan ahead you can make a widget, set its attributes, and then Copy and Paste. The pasted elements retain the attributes of the original. You can also use middle mouse drag and drop while holding down the Control key to perform a quick copy.

- Can I select multiple widgets?

  No, RapidApp does not currently support this. Builder Xcessory, the full-featured program on which RapidApp is based, does support multiple selected widgets.

- How do I align 2 or more widgets?

  The best way to align widgets in IRIS IM is to select the appropriate layout containers. The RowColumn widget is often used for this. You should also look at the **SgSpringBox** widget, which provide easy and powerful control over alignment of its children.

- Why doesn't RapidApp provide some sort of alignment tool for arranging widgets?

  Alignment tools only make sense when applied to multiple widgets, and RapidApp cannot select multiple widgets at once.

- How do I create layouts that resize?

  It depends on the layout. The most frequent approach is to use a Form widget and specify attachments to determine how each child resizes. The behavior of the Form widget is complex and you should consult a Motif book for more information.

  There are other possibilities. The Rubber Board is easy to use, but has some limitations. The Spring Box offers a resizable layout that is the entire basis of some toolkits. The Paned Window offers a limited form of resizable layout. The RowColumn also offers limited resizability when its **adjustLast** resource is set to true.

Generally, programs use a combination of all these widgets.

- Why is it so hard to create resizable layouts? Isn’t this trivial on PCs?

  Second question first: No. Most PC packages offer only a Bulletin Board container. All elements are arranged according to specific positions and never change. If a programmer wants to support resizing, she handles a resize event on the Bulletin Board widget and programmatically recomputes the layout of each and every element in the container and resets the position.

  You can do exactly that, if you wish, by simply choosing a Bulletin Board and adding an event handle for StructureNotify events. Then you are on your own, just like in the PC environments.

  IRIS IM makes it much simpler to achieve resizable layouts by providing containers with their own built-in algorithms for rearranging children when the container is resized. This greatly simplifies the task, and eliminates the need to write a great deal of error-prone code yourself. However, it does make it necessary to understand the algorithm supported by each container, along with the various options that can be used to alter the algorithm, and to be able to choose the algorithm you want to apply.

  The difference is much like the choice of using a library of reusable classes vs. writing your own. Writing your own classes for hash tables, etc., is straight-forward and you can get started right away, but you have to do all the time-consuming work. If you try to use a class from a library, you may have a learning curve that keeps you from writing productive code until you understand what the existing class does. But once you understand, you can use the class over and over without having to rewrite the code from scratch each time.

- Why are the entries under the Project menu grayed out?

  These entries invoke other Developer Magic tools, and are available if you have the tools installed. You will not be abel to build, debug, or run, for example if you do not have *cvbuild* installed. The Browse Source entry invokes *cvstatic*, which must be installed. In addition, for this entry, you must check “Use Cvstatic” in the Application options dialog. When active, the makefile automatically generates a static analysis database. While useful, this greatly slows down the compilation, so the option is off by default.

- How can I change a class after I have created it?

  You need to switch to class edit mode, using the option under the Classes menu pane. In this mode, you see a single copy of each class you have created. Any changes apply to all instances of that class.

- Once I've defined a class, how do I get the object into an existing container?

  When you create a class you are originally working on a collection of widgets, which becomes an instance of the class. Once a class has been defined, you can delete the instance if you wish. New instances of the class can be created from the "User Defined Classes" palette at any time. If you would like to just use the existing instance, but place it in some other container, simply drag the object into the container using the middle mouse button.

- How do I hook up a Help System to the programs RapidApp creates?

  Your program calls into a specific API for requesting help. There are several libraries that can supply that API, and you can also write your own. The *vkhelp* library distributed with ViewKit provide a simple help system that posts dialogs based on X resources to provide simple help. The source to this library is also part of the ViewKit examples, as a starting point for writing your own help system.

  The best option, though, is to use the Insight-based SGI Help System. See the Indigo Magic Integration Guide for details on how to use this system.

- Why does it take so long to create some entries in the ViewKit or Inventor palettes?

  Some of the elements on these and other palettes are dynamically loaded from shared libraries, only when needed. The first time one of these libraries is loaded, the symbols in that library must be resolved before proceeding. This operation may cause a slight delay.

- Can I add my own widgets to RapidApp?

  No, RapidApp does not support the addition of custom widgets at this time. You can add your own widgets to Builder Xcessory, so you may want to consider an upgrade if you need to add custom widgets.

- Can I add the C++ classes I created back onto the RapidApp palette?

  You can save a file of components to be loaded into RapidApp for later use. To do this, define your classes, delete all instances, and save the file. Only the class descriptions will be saved. You can then use the Import command on the File menu to load these classes.

- I want to add C++ classes to RapidApp, but I'd like to add functionality to the classes and see that functionality reflected in the classes created with RapidApp, like the ViewKit and Inventor classes. How can I do this?

  This is technically possible, but is not supported in the current release of RapidApp. A clean mechanism for adding classes in this manner may be added in a future version.

- RapidApp creates ViewKit programs. How do I get ViewKit?

  ViewKit is bundled with the Silicon Graphics C++ product. If you have C++, you already have ViewKit.

- How can I find out more about ViewKit? Is there any documentation?

  The ViewKit Programmer's Guide is available on-line if you install ViewKit. You can also order a hard-copy of the manual. Man pages are also part of the ViewKit distribution.

- How portable is the code generated by RapidApp?

  It depends on what elements you use. If you stick to standard IRIS IM widgets, then the dependencies are X, Xt, IRIS IM, and ViewKit. So, assuming ViewKit is available on the platform(s) of interest, there should be no problem. The Makefiles, desktop icons, ftr rules, and so on are specific to SGI, of course.

  If you use Inventor classes, you are limited by the availability of Inventor on your target platform. The same goes for other C++ components that may be available for RapidApp from SGI now or in the future.

Several widgets are unique to SGI, including the OpenGL widget, the Rubber Board, Spring Box, Thumb Wheel, Dial, Finder, and so on. If you use these widgets, you may not directly port to other platforms.

- I want to use C (Ada, Fortran, Pascal, Cobol, ASM, TCL,...), can RapidApp generate these languages instead of C++?

  RapidApp is designed to help you write object-oriented programs using the Silicon Graphics C++ class libraries, and derives a great deal of its power from the use of object technology and the underlying libraries. While support for other languages may be theoretically possible, only C++ is available at this time. This could change in the future.

  C support is also available by upgrading to Builder Xcessory, although you will not enjoy the degree of automatic support for the Indigo Magic Desktop environment when generating C code using Builder Xcessory.

- I'd like to add TCL support to my programs. Can RapidApp help?

  Not presently, although support for integrating TCL or similar scripting languages into ViewKit programs may be supported in the future. There is nothing to prevent you from creating TCL interpreters, or otherwise integrating TCL into your programs yourself, of course.

- How do I set colors for my widgets?

  RapidApp is designed to strongly support SGI-style applications, and the look and feel of the SGI desktop is highly dependent on "schemes". By default, RapidApp creates programs whose colors are determined solely by schemes. You can always programmatically set colors for widgets. All widgets are available to derived classes as protected data members, so it is easy to override colors. You can also set colors in your app-defaults file. Simple specify the class name of your program as part of the resource, to override the scheme settings.

  If you simply don't like the scheme being applied, you can change the scheme your programs use with the scheme browser, available from the Toolchest. Also see Chapter 3, "Windows in the Indigo Magic Environment," in the *Indigo Magic User Interface Guidelines* and Chapter 3, "Using Schemes," in the *Indigo Magic Desktop Integration Guide* for information on schemes.

- How do I change the font used by elements of my program.

  Like colors, fonts in the Indigo Magic Desktop environment, are controlled by schemes. You can always override either programmatically, or in your app-defaults file, if needed. However, you are encouraged to use the scheme-specified fonts. There are some cases where it makes sense to change the font, particularly labels, text and lists. RapidApp allows you to switch the font on these items to one of the scheme-supported fonts. For example, you might want to change a label widget from bold to normal, or a text widget to a fixed width font.

- How can I see the full set of resources supported by IRIS IM?

  RapidApp limits the available resources to those most commonly useful to programmers. You can always set any resource in your program or in the app-defaults file. You can always determine the full set of resources by looking at the man pages for the IRIS IM widget of interest, or consulting a IRIS IM reference guide.

- How can I browse the Tips RapidApp displays at startup?

  Currently, you cannot. But all the information, and more, is available in the RapidApp User's Guide.

- How can I get rid of the Tips show at startup?

  Turn them off from the Preferences panel, found under the options menu.

- Is there an easy way to add a company standard header to my files?

  Yes, set the X resource **userHeaderComments** to the name of a file to be inserted. The file should be properly formatted, with C++ comment characters. For example:

  ```
  *userHeaderComments: /usr/include/StandardLegalNotice
  ```

  would cause the file */usr/include/StandardLegalNotice* to be inserted into each file generated by RapidApp before any other comments or code created by RapidApp.

- RapidApp-generated code uses leading underscores for data member names. Isn't that illegal? Why is this?

  It is not at all illegal, even Bjarne Stroustrup's C++ reference demonstrates this technique. All protected data members are given leading underscores, partly as an indication that these variables are protected members (leading underscores have long been used to denote "private") and also to prevent name collisions with member functions of the same name. For example, you can have a data member _name and an access function name() that retrieves _name. This is just a convention used throughout ViewKit, and happens to lend itself well to code generation as well.

- Sometimes I enter a name in RapidApp and RapidApp adds a number after it. (like "label" becomes "label1"). Why is this and how can I avoid it?

  The UIL format used as the underlying document model for RapidApp, as well as many other interface builders, requires unique names for all symbols. RapidApp enforces this convention wen you enter the names, rather than causing an error later. You can avoid this behavior by adopting conventions for your user interface elements. For example, an "ok" button, might have the label "OK", the name "okButton" and a callback function "ok()".

- How do I provide client data to my callbacks?

  Client data is meaningless in the style of code generated by RapidApp. In C, it is necessary to pass state around to callbacks via clientData. In C++, the functions called as a result of an action are member functions, and all the state that can be accessed is available within the current object. Anything else that you could possibly pass as clientdata would be within another class, and therefore passing it in some way would be a violation of encapsulation. The only remaining use of client data would be to pass some simple value, such as an integer code or string to provide information about the context of the call. While occasionally useful, there are other ways to deal with this need (different "callbacks" that call a second function, for example).

- How do I add dynamic behavior to classes created using RapidApp?

  The best way is simplest: just use your favorite editor to add data members, member functions, etc. Remember that classes are created in pairs, base class ("UI") and derived classes. RapidApp always uses the derived class, so you can add all new code to the derived class and avoid modifying the base class. The merge feature of RapidApp will assure that your changes are maintained.

- How do I connect classes created using RapidApp to each other?

  Connecting classes in C++ is always challenging because of C++'s strong typing. There are two basic ways to do this that work well with RapidApp-generated (ViewKit-style) classes. The first is to hard code the connection by implementing an API that each class can use to connect to the other as needed. The second is to use the ViewKit support for callbacks.

  Say you have two classes, ***Input*** and ***Output***. ***Input*** has a text field and you would like ***Output*** to be notified when the user enters text in ***Input***'s text field. You could use either of the following approaches:

  – Hard coding the connection, using your favorite editor:

    Add a member function ***newText()*** to class ***Output***. This member function will do whatever it is you want to do when new text is available.

    Add a public member function ***void setOutput(class Output *)*** to *Input.h*

    Add a private or protected data member class Output* _output to *Input.h*

    Add #include "Output.h" in *Input.C*.

    Implement setOutput(* output) { _output = output; } in *Input.C*

    At the point where you know text is entered (an **activateCallback**) in *Input.C*, call _output->newtext();

  – Using ViewKit callbacks:

    Add a private or protected member function to Output:

```
void Output::textEntered(VkCallbackObject *, void *, void *);
```

    Somewhere do:

```
VkAddCallbackMethod(Input::newText, input, output, &Output::textEntered, NULL);
```

    where input is the instance of Input and output is the instance of Output

    Then in *Input.h*, add a static public member:

```
const char const * newText;
```

    Then in *Input.C*, declare the static member:

```
const char const *Input::newText = “newText”;
```

At the point where you know text is entered (an **activateCallback**) in ***Input***, call

```
callCallbacks(newText, NULL);
```

Output::textEntered() will be invoked.

- How do I access the widgets inside a component?

  In general, you should not. A class is (or should be) a class because it represents an abstraction. The details are encapsulated in the class. A class is not merely a collection of widgets. You should think of the class as an entity in its own right and design the API of the class independent of its implementation.

  For example, assume you would like to change a label in a class to “red” to indicate an error condition has occurred. You *could* write an access function for the label widget and use ***XtSetValues()***, etc., to change the color, but this would be a flagrant violation of encapsulation and object-oriented design. Specifically, the internal details of your implementation (that you have a specific label widget whose color can be set directly) have now become part of your public API.

  A better approach would be to write a public member function, perhaps:

  ```
  void setStatus(Status);
  ```

  where Status is a type that includes Error, Warning, Normal, and so on. What exactly happens in that ***setStatus()*** method is now up to the class. You could set the label widget to red, for example. Later you could decide to change the label to a 3D viewer, and sound an audible alarm when an error occurs without breaking other classes that depend on the public API.

Appendix B

RapidApp Reference

This appendix describes in the function of each window, menu, widget, and display in the RapidApp's graphical user interface (GUI). This appendix contains the following sections:

- "Global Objects"
- "Windows Palette"
- "Containers Palette"
- "Controls Palette"
- "Menus Palette"
- "ViewKit Palette"
- "Inventor Palette"

RapidApp consists of several palettes, each palette containing several user-interface elements. Each palette and its interface elements are described in detail in their own sections in this chapter.

Global Objects

This section describes RapidApp's global objects—the objects that are common from palette to palette. These objects include the menu bar items and the palette tabs (see Figure B-1).

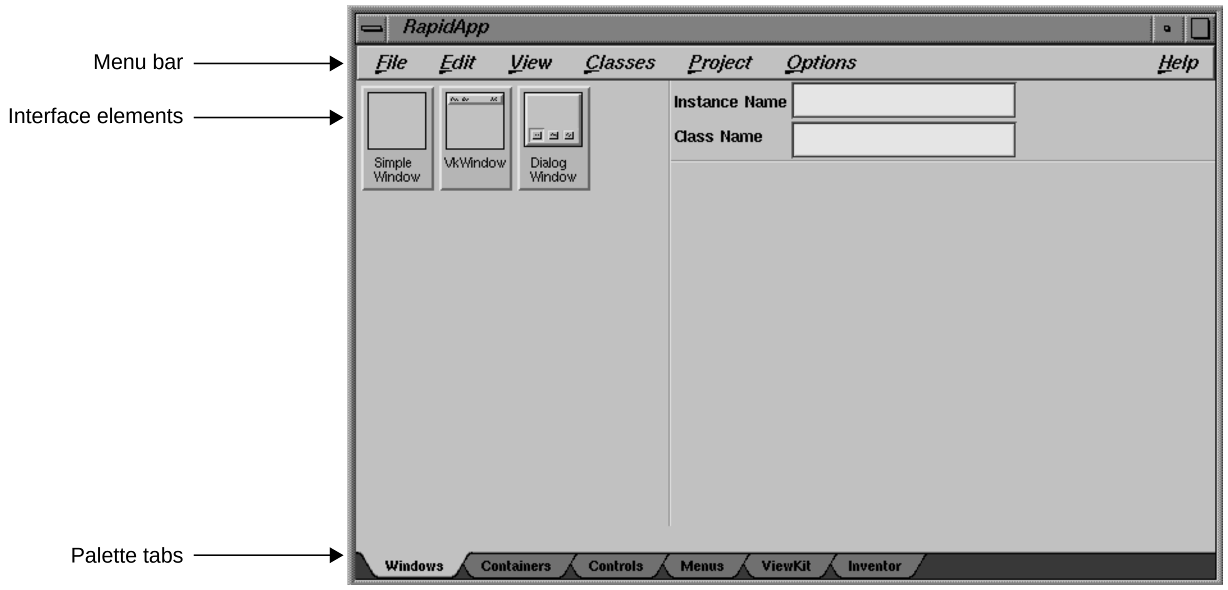


Figure B-1 RapidApp Window

File Menu

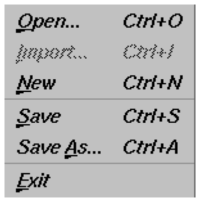


Figure B-2 File Menu

The File menu (see Figure B-2) allows you to save and open RapidApp files. You can also quit RapidApp through the File menu. The File menu contains the following selections:

Open...	Displays the Open File dialog to allow you to open a file.
Import...	Displays the Open File dialog to allow you to import a file. RapidApp adds the contents of the file to the current interface.
New	Clears the current interface in preparation for creating a new interface. RapidApp gives you the option of retaining the current user-defined components.
Save	Saves your current session to a file. If you haven't provided a filename previously, RapidApp uses the default filename *save.uil*.

Save As...	Displays the Save File dialog, which allows you to save your current session to a file with a filename of your choice.
Exit	Exits RapidApp.

Edit Menu

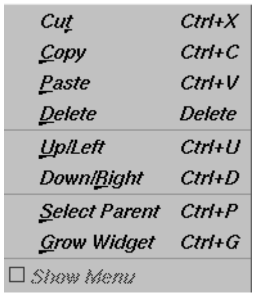


Figure B-3 Edit Menu

The Edit menu (see Figure B-3) supports cut, copy, and paste operations, as well as some commands for manipulating a selected interface element. The Edit menu contains the following selections:

Cut	Cuts the currently selected element (and, if it’s a container, all of its children) and place the element on the clipboard.
Copy	Copies the currently selected element (and, if it’s a container, all of its children) to the clipboard.
Paste	Pastes the element currently on the clipboard (and, if it’s a container, all of its children) into your interface.
Delete	Deletes the currently selected element (and, if it’s a container, all of its children). This option **doesn’t** place the element on the clipboard.
Up/Left	In those containers where the creation order of its child elements determines their position, moves the currently selected child one position up or left.
Down/Right	In those containers where the creation order of its child elements determines their position, moves the currently selected child one position down or right.
Select Parent	Selects the parent of the currently selected element.
Grow Widget	Increases the horizontal and vertical size of the selected element by 20 pixels.
Show Menu	If the currently selected element is a menu cascade button, displays or hides its corresponding menu pane.

View Menu

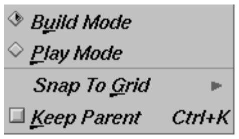


Figure B-4 View Menu

The View menu (see Figure B-4) controls the build / play mode selection as well as determining constraints for placing elements. The View menu contains the following selections:

Build Mode	Enables build mode, the mode you must be in when creating a new application. Toggling on build mode toggles off the Play Mode toggle.
Play Mode	Enables play mode, which allows you to run your application to check its functionality. Toggling on play mode toggles off the Build Mode toggle.
Snap To Grid submenu (see Figure B-5)	Allows you to set your snap to grid value to one of five settings through a list of toggles: Off, 2, 5, 10, and 20.
Keep Parent	Toggles explicit selection mode. When on, RapidApp limits selection of new elements to those accepted by the currently selected element. Also, new elements that you create are added to the selected element instead of the container on which you drop them.

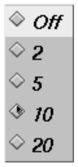


Figure B-5 Snap To Grid Toggles

Classes Menu

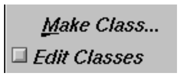


Figure B-6 Classes Menu

The Classes menu (see Figure B-6) allows you to create and edit user-defined components. The Classes menu contains the following selections:

Make Class...	Displays the Make Class dialog. This dialog allows you to convert the currently selected element and all of its children into a C++ class.
Edit Classes	When toggled on, RapidApp hides your current interface and displays all user-defined components. You can then select, edit, and manipulate the individual elements composing the classes.

Project Menu

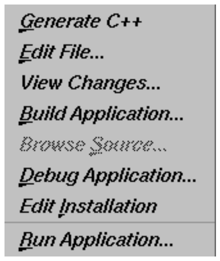


Figure B-7 Project Menu

The Project menu (see Figure B-7) allows you to generate code, browse and edit files, build an application, run the program under a debugger, and so on. The Project menu contains the following selections:

Generate C++
: Converts the application that you created with RapidApp into C++ code.

Edit File...
: Displays the Edit File dialog, which allows you to open and edit a file.

View Changes...
: Displays the Select File to Compare dialog, which allows you to select a file and compare it to the previously saved version. You can also use this option to manually merge changes if you want.

Build Application...
: Launches the Developer Magic Build Manager. If you are currently using the debugger, the executable will automatically be detached from the debugger and re-attached when the compilation is completed.

Browse Source...
: Launches the Static Analyzer to analyze the structure of your application. To use this option, you first must create a static analysis fileset and database for your application.

Debug Application...
: Launches the Developer Magic Debugger. If your application isn't up-to-date, RapidApp automatically invokes the Build Manager to update the executable.

Edit Installation
: Launches Software Packager, a graphical tool for creating and editing installable images.

Run Application...
: Runs your application. If your application isn't up-to-date, RapidApp automatically invokes the Build Manager to update the executable.

Options Menu

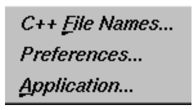


Figure B-8 Options Menu

The Options menu (see Figure B-8) allows you to set several options related to RapidApp operation and code generation. The Options menu contains the following selections:

C++ File Names...
: Displays the C++ File Options dialog (see "C++ File Options Dialog"), which allows you to specify C++ file extensions and the name for your makefile.

RapidApp Preferences...
: Displays the RapidApp Preferences dialog (see "RapidApp Preferences Dialog"), which allows you to set preferences controlling RapidApp operation.

Application...
: Displays the Output Application Names dialog (see "Application Names Dialog"), which allows you to specify application file and class names and various application characteristics.

The dialogs are discussed in the following sections.

C++ File Options Dialog

The C++ File Options dialog allows you to customize the source and fil extensions for your files as well as to control the Makefile generation for you;r program. The source and header suffixes can be any valid suffix supported by the C++ compiler and SGI's makefiles. The accepted file extensions are the following: *.c*, *.C*, *.cxx*, *.c++* and *.h* for headers.

Because makefiles often need to grow into complex files, RapidApp allows you to turn off the generation of a Makefile completely. This allows you to write your own makefile without any need to merge changes made in RapidApp. Note that if you turn off makefile generation, new files are not added; you are responsible for the makefile creation and maintenance. You can also choose to use a different name for the generated makefile so as to avoid overwriting a custom makefile, while still having the convenience of the generated makefile when needed.

RapidApp Preferences Dialog

The RapidApp Preferences dialog allows you to customize the behavior of RapidApp itself. The following toggle options are available:

Automatically Dismiss Startup Screen
: The startup screen will disappear as soon as the program is ready to run without the ned to manually dismiss it.

Show Tips at Startup
: If true, show a random tip about how to use RapidApp on the startup screen. If false, no tip is shown, and the panel will also be dismissed automatically.

Enable Sound
: If true, RapidApp uses soundscheme to provide audio feedback for various operations. You can also disable sound from the desktop control panel on the ToolChest.

Use $WINEDITOR to edit files
: If True, the Edit Files... entry on the Project menu will launch the edit indicated by the WINEDITOR environment variable. If false, RapidApp launches the CaseVision source editor/viewe

Confirm before deleting shells from window manager
: If True, RapidApp will post a warning dialog if you attempt to dismiss a top-level user interface element using the window manager close control.

Confirm before deleting containers
: If true, RapidApp will post a warnignng dialog whenever you delete a container widget, to avoid accidental deletions of elements that might be hard to reconstruct.

Application Names Dialog

The Application dialog controls various code generation options that affect the way an application behaves or is built. Most of these options do not take effect until the next time code is generated.

The following options are available:

Directory Path
: This controls the directory in which the application source will be placed when generated. If the directory does not exist, RapidApp will prompt the user to determine whether or not to attempt to create it.

Application Name
: The name of the program to be created.

Application Class
: The X application class. Following X conventions, this should be the name of the application with the first letter capitalized.

Use Tooltalk
: If true, the application will support basic tooltalk communication using the ViewKit VkMsg facility.

Use Runonce
: If true, the program will use the VkRunOnce facility, which ensures that only one instance of the application is running at any one time. See the VkRunOnce man page for details.

Strip Sizes from generated code
: If true, RapidApp will remove all hard coded sizes from the generated code. RapdiApp sometimes sets widths and heights more aggressively than it should, and this option allows you to remove all such sizes. Note that this could break your layout, unless you are sure your layout does not depend on any widths or heights.

Don't Merge generated code
: If true, RapidApp will nto attempt to merge any files when code is generated. Instead, any files that differ from the current files will be renamed for you to manually integrate.

License-protect Application
: If true, the application will include the code to setup NETLS license server. See the VkNLS man page for details.

Generate Windows Automatically
: If true, any user interface element that is in a toplevel shell will be treated as a child of a VkSimpleWindow. Setting this to false allows the creation of stand-alone classes that are not created as part of any top-level window.

Use VkEZ Convenicne Library
: If true, code will be generated for the headers and libraries of the **VkEZ** facility.

Palette Tabs

You access the different RapidApp palettes through the palette tabs (see Figure B-9) at the bottom of the RapidApp window. Click a tab to display the corresponding palette.

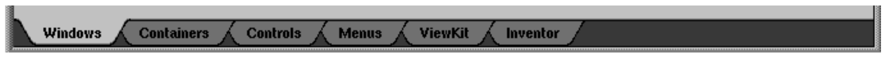


Figure B-9 Palette Tabs

Keys and Shortcuts

This section describes the accelarator keys available in RapidApp.

Shift+F1	Prompts for click for help.
Ctrl+O	Opens a file.
Ctrl+I	Imports a file.
Ctrl+N	Starts a new project, deleting all current elements.
Ctrl+S	Save a file.
Ctrl+A	Save a file as a new name.
Ctrl+P	Select the parent of the currently selected element.
Shift-Ctrl-Left mouse	Select the parent of the currently selected element.
Click on selected menu	Displays the menu pane.
Ctrl+G	Increases the size of the currently selected element.
Ctrl+X	Cuts the currently selected element to the clipboard.
Ctrl+C	Copies the currently selected element to the clipboard.

Ctrl+V	Pastes the contents of the clipboard.
Delete	Deletes the curently selected element without placing it on the clipboard.
Backspace	Deletes the curently selected element without placing it on the clipboard.
Ctrl+U	Repositions a widget inside a RowColumn widget, moving it up, if the parent's orientation is vertical.
Ctrl+D	Repositions a widget inside a RowColumn widget, moving it down, if the parent's orientation is vertical.
Ctrl+K	Toggle Keep Parent mode.
Arrow keys	Moves an element in the corresponding direction.
Left Mouse Button	Selects an element and moves it within the same container.
Middle Mouse Button	Drags an object between containers.
Ctrl-Left Mouse Button	Maintains the currently selected element.
Drag and Drop from Desktop	Bitmaps, Pixmaps, and various other files can be dragged from the Indigo Magic desktop directly onto various widgets to set the associated resource.

In a child of a Form, the following accelarators are enabled:

- Right mouse button over an attachment icon pops up a menu of attachments
- Shift-left mouse over an attachment icon adjusts the offset
- Left mouse button over an offset drags the attachment to another location

Windows Palette

The Windows palette (see Figure B-10) contains window interface elements.

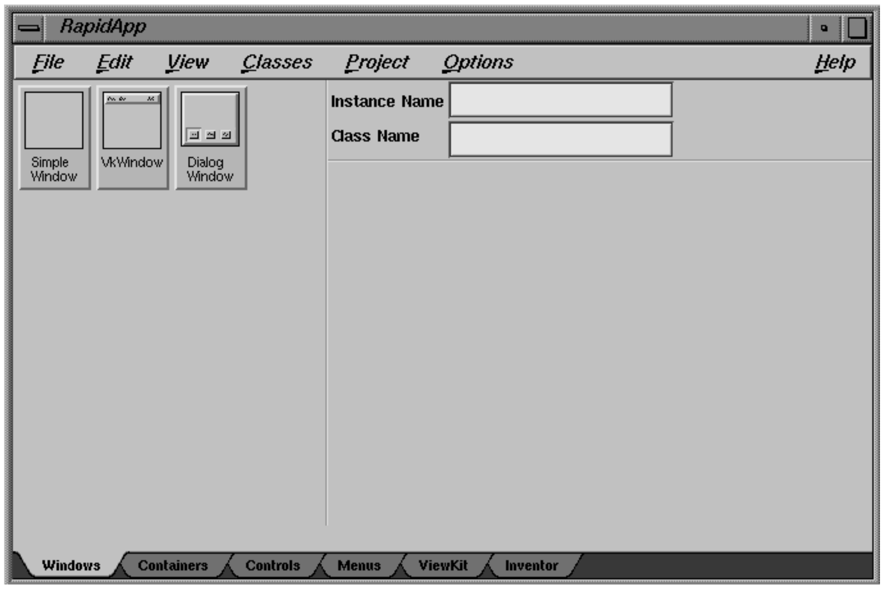


Figure B-10 Windows Palette

The user interface elements available through this palette are described in the following sections.

VkSimpleWindow

The **VkSimpleWindow** class implements a simple top-level window to be used by IRIS ViewKit applications. Use **VkSimpleWindow** when you don't want a menu bar.

VkSimpleWindow Resources

Following are the **VkSimpleWindow** resources:

coprimaryWindow
If this resource is set to True, this window is treated as a co-primary (secondary) window as defined by the SGI Style guide. The window will not be created by default on startup, and the application is responsible for creating and displaying it when it is needed.

disableIconify If this resource is set to True, the user's ability to iconify the window will be disabled.

disableWindowResize
If this resource is set to True, the user's ability to resize the window will be disabled.

VkWindow

The **VkWindow** class behaves similarly to **VkSimpleWindow** except that it provides additional support for a menu bar, based on the **VkMenuBar** class and related **VkMenuItem** classes.

VkDialogWindow

The **VkDialogWindow** provides a top-level dialog window for constructingcustom dialogs that conform to the API provided by the **VkDialogManagerclass**. To create a dialog, add a single container and then populate that container with the interface of your choice. The container you place in the dialog window should represent a class, and is forced to be a class if you do not explicitly make it so. This class automatically contains the **ok()**, **cancel()**, and **apply()** member functions, which are called as needed when the user interacts with the dialog.

The actual buttons displayed by the dialog are determined dynamically as with all **VkDialogManager** subclasses.

Dialogs can be posted programmatically by calling **post()**,**postBlocked()**, **postModal()**, or **postAndWait()**. See the **VkDialogManager** reference page for more information.

Containers Palette

The Containers palette (see Figure B-11) includes container interface elements such as bulletin boards and radio button boxes.

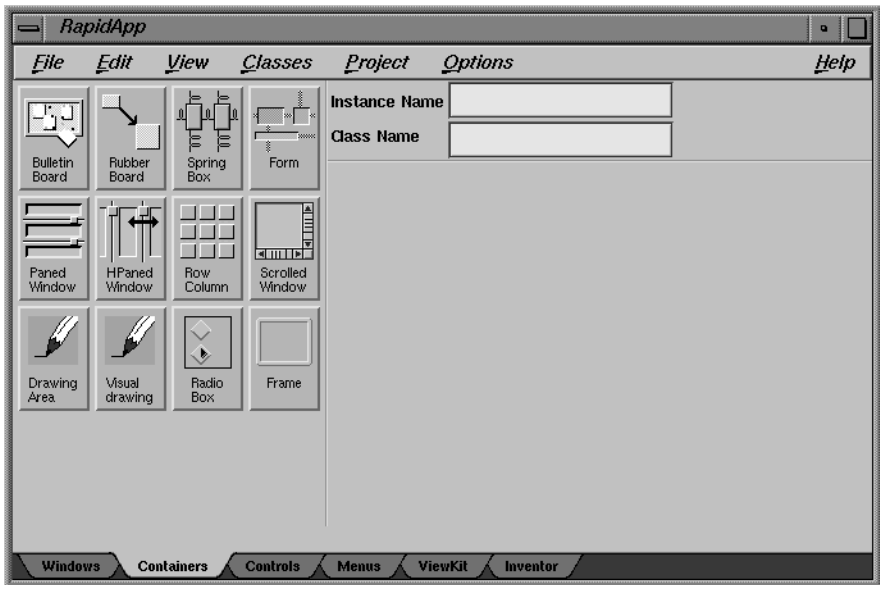


Figure B-11 Containers Palette

The user interface elements available through this palette are described in the following sections.

BulletinBoard

The **BulletinBoard** widget is a container widget that has no layout algorithm. The location and size of each child is based solely on where and how the child is placed using RapidApp. Layouts based on the **BulletinBoard** widget cannot be resized and do not respond to changes to individual widgets.

BulletinBoard layouts are not appropriate for programs that will be customized or internationalized. The **BulletinBoard** widget is most suitable for beginners and for quick prototypes.

BulletinBoard Resources

Following are the **BulletinBoard** resources:

XmNmarginHeight
Specifies the minimum spacing in pixels between the top or bottom edge of **BulletinBoard** and any child widget. You must be careful when positioning children using RapidApp, because the **BulletinBoard** enforces this margin only at creation time. The **BulletinBoard** allows you to use RapidApp to place children in the margin area interactively. However, when the children are initially created in the final program, the **BulletinBoard** moves the children out of the margin area when the child is initially created.

XmNmarginWidth
Specifies the minimum spacing in pixels between the left or right edge of **BulletinBoard** and any child widget. The same restrictions apply as in the **XmNmarginHeight** resource.

SpringBox

The **SgSpringBox** widget is a container widget that arranges its children in a single row or column based on a set of spring resources associated with the child. The **SgSpringBox** widget allows layouts similar to those supported by the **XmForm** widget, but is sometimes easier to set up and allows you to create some layouts that cannot be achieved with the **XmForm** widget. For example, centering a column of widgets is very easy to do with the **SgSpringBox** widget, but nearly impossible using the **XmForm**.

Each child of an **SgSpringBox** widget has the following constraints associated with it:

- Each child has a “springiness” in both the vertical and horizontal direction that determines how much the child may be resized in each direction. The **XmNverticalSpring** and **XmNhorizontalSpring** resources control the degree of "springiness" in each child. A value of zero means the child cannot be resized in that direction. For non-zero values, the values are compared to the values of other springs in the overall system to determine the proportional effects of any resizing. The default value of both resources is zero.

- Each child also has a spring between its left, right, top, and bottom sides and whatever boundary it is adjacent to. The value of any spring resource can be altered in RapidApp's resource editor. Selecting any child displays its resources in the Resource Editor window.

Several common default layouts can be created using the **XmNdefaultVerticalLayout** and **XmNdefaultHorizontalLayout** resources supported by the **SpringBox**. More complex layouts can be achieved by editing the constraint resources of the individual children.

SpringBox Resources

Following are the **SpringBox** resources:

XmNmarginHeight
Specifies the minimum spacing in pixels between the top and bottom edges of the **SpringBox** and any child widget.

XmNmarginWidth
Specifies the minimum spacing in pixels between the left or right edge of the **SpringBox** and any child widget.

XmNminSpacing
Specifies the minimum spacing between the children of the **SpringBox**.

XmNorientation
The **XmNorientation** resource determines whether the **SpringBox** is vertical or horizontal. If you change this resource after children have been added, you may have to reset individual spring values for the new layout. The existing resources retain their current values when orientation changes. No attempt is made to map existing settings to account for the orientation.

XmNdefaultVerticalLayout, **XmNdefaultHorizontalLayout**
These resources provide a convenient way to apply a collection of resource settings to all current children of a

SpringBox. Each resource is independent and controls only the resources that apply vertically or horizontally. The meaning of these resources does not change with the **XmNorientation** resource. That is, vertical is always vertical. Possible layouts include:

- **XmCENTER**: Centers all children in the middle of the **SpringBox**, with equal spacing on either side of the entire group of children.
- **XmSPAN**: Stretches all children equally to fill the entire space of the **SpringBox**.
- **XmLEFT**: Sets all children to their natural size and moves to the left edge of the **SpringBox**.Only applies to **XmNdefaultHorizontalLayout**.
- **XmRIGHT**: Sets all children to their natural size and moves to the right edge of the **SpringBox**.Only applies to **XmNdefaultHorizontalLayout**.
- **XmTOP**: Sets all children to their natural size and moves to the top edge of the **SpringBox**.Only applies to **XmNdefaultVerticalLayout**.
- **XmBOTTOM**: Sets all children to their natural size and moves to the bottom edge of the **SpringBox**.Only applies to **XmNdefaultVerticalLayout**.
- **XmDISTRIBUTE**: Sets all children to their natural size and distributes them evenly across any open space in the **SpringBox**.
- **XmSTRETCH_FIRST**: Allows the first (left-most or top-most) child to stretch freely to fill any available space. All others are set to their natural size.
- **XmSTRETCH_LAST**: Allows the last (right-most or bottom-most) child to stretch freely to fill any available space. All others are set to their natural size.
- **XmIGNORE**: Ignores the default setting and uses the custom values of each individual widget's spring resources

SpringBox Constraint Resources

Following are constraint resources that are added to children of a **SpringBox** widget. These resources determine the stretchability of the space adjacent to the associated side of the widget. The larger the value, the more this space can be resized relative to other “springs” contained in the **SgSpringBox** widget.

XNleftSpring	Sets the relative springiness of the space to the left of the widget.
XmNrightSpring	Sets the relative springiness of the space to the right of the widget.
XmNtopSpring	Sets the relative springiness of the space above the widget.
XmNbottomSpring	Sets the relative springiness of the space below the widget.
XmNverticalSpring	Sets the relative springiness of the widget in the vertical direction
XmNhorizontalSpring	Sets the relative springiness of the widget in the horizontal direction

Form

The **Form** is a container widget that arranges its children based on constraint resources associated with each child. Resources supported by each child of the **Form** define attachments for each of the child’s four sides. These attachments can be to the **Form**, another child widget or gadget, a relative position within the **Form**, or the initial position of the child. The attachments determine the layout behavior of the **Form** when resizing occurs.

Attachments are made in RapidApp directly on each child of a **Form**. Each **Form** child has small attachment handles on each of its four sides. These attachment handles support several operations:

Left mouse button

You can click the left mouse button on an attachment handle and drag an attachment from the current widget to any other widget, including its parent (the **Form**). This indicates either an **XmATTACH_WIDGET** (see "Form Resources") or **XmATTACH_FORM** value for the attachment.

Right mouse button

Posts a menu that allows you to choose from the various attachment types for each side.

Shift left mouse button

Posts a menu that shows the current offset for an attachment. Moving the mouse while holding down **`<Shift>`**-left-button changes the offset.

Form Resources

The following resource affects the behavior of the **Form** widget itself.

XmNfractionBase

Specifies the denominator used in calculating the relative position of a child widget that uses an **XmATTACH_POSITION** attachment. The value must not be 0.

If the value of a child's attachment resource is **XmATTACH_POSITION**, the position of the corresponding side of the child is relative to the left (or top) side of the **Form** and is a fraction of the width (or height) of the **Form**. This fraction is the value of the child's position resource divided by the value of the **Form**'s **XmNfractionBase**.

Form Constraint Resources

These resources are supported by all children of a **Form** widget.

XmNbottomAttachment, **XmNtopAttachment**, **XmNleftAttachment**, **XmNrightAttachment**

These resources specify the attachment of the bottom, top, left, or right sides of the child. Each resource can have the following values, which can be selected from a popup menu posted by pressing the right mouse button over the bottom attachment icon:

- **XmATTACH_NONE**: Do not attach this side of the child.
- **XmATTACH_FORM**: Attach this side of the child to the near side of its parent.
- **XmATTACH_OPPOSITE_FORM**: Attach this side of the child to the far side of its parent. The corresponding offset resource also affects the final position of the child.
- **XmATTACH_WIDGET**: Attach this side of the child to the near side of another widget. Normally, **XmATTACH_WIDGET** is specified by pressing the left mouse button over the attachment icon and dragging out the attachment to the desired widget. Once an attachment is made, the popup menu can be used to switch between **XmATTACH_WIDGET** and **XmATTACH_OPPOSITE_WIDGET**. The corresponding offset resource also affects the final position of the child.
- **XmATTACH_OPPOSITE_WIDGET**: Attach this side of the child to the far side of another widget. The corresponding offset resource also affects the final position of the child.
- **XmATTACH_POSITION**: Attach this side of the child to a position that is relative to the left (or top) side of the **Form** and in proportion to the width (or height) of the **Form**. The actual position is determined by the **XmNbottomPosition**, **XmNtopPosition**, **XmNleftPosition**, or **XmNrightPosition** resources in conjunction with the **XmNfractionBase** resource. The corresponding offset resource also affects the final position of the child.

XmNBottomOffset, **XmNtopOffset**, **XmNleftOffset**, **XmNrightOffset**
Specifies the constant offset between the corresponding side of the child and the object to which it is attached. The

relationship established remains, regardless of any resizing operations. RapidApp allows you to enter this value in the resource editor, alter the value by repositioning the child, or change the value by holding down the Shift key while pressing the left mouse button over an attachment icon and dragging the pointer. In the last case, the current offset value is displayed in a popup menu during the drag. In general, moving or resizing a child of a form in RapidApp corresponds to changing the value of one or more offsets, and only indirectly the position or size.

XmNtopPosition, **XmNbottomPosition**, **XmNleftPosition**, **XmNrightPosition**
Determines the position of the corresponding side of the child when the corresponding attachment is set to **XmATTACH_POSITION**. In this case the position of the side of the child is relative to the left (or top) side of the **Form** and is a fraction of the height of the **Form**. This fraction is the value of the child's position resource divided by the value of the **Form**'s **XmNfractionBase**. For example, if the child's **XmNbottomPosition** is 35, the **Form**'s **XmNfractionBase** is 100, and the **Form**'s height is 200, the position of the bottom side of the child is 70.

RowColumn

The **RowColumn** widget is a general-purpose **RowColumn** manager capable of containing any widget type as a child. The type of layout enforced by the **RowColumn** is controlled by how the application has set the various layout resources. It can be configured to lay out its children in either rows or columns. In addition, the application can specify that the children be laid out as follows:

- the children are packed tightly together into either rows or columns
- each child is placed in an identically sized box (producing a symmetrical look)
- a specific layout (the current X and Y positions of the children control their location)

RowColumn Resources

Following are the **RowColumn** resources:

XmNadjustLast
: If **XmNadjustLast** is set to true, the last row of children is stretched to fill the **RowColumn** to the bottom edge when **XmNorientation** is **XmHORIZONTAL**. The last column of children is extended to the right edge of **RowColumn** when **XmNorientation** is **XmVERTICAL**.

XmNentryAlignment
: This resource controls the alignment type for children that are subclasses of **XmLabel** or **XmLabelGadget** when **XmNisAligned** is set to true. These are the possible alignment values:

- **XmALIGNMENT_BEGINNING**
- **XmALIGNMENT_CENTER**
- **XmALIGNMENT_END**

XmNisAligned Specifies text alignment for each **XmLabel** (or subclass) child of a **RowColumn** widget. The **XmNentryAlignment** resource controls the type of textual alignment.

XmNnumColumns
: Specifies the number of rows or columns supported by the **RowColumn** widget. The resource controls the number of elements in the minor dimension; this resource is meaningful only when **XmNpacking** is set to **XmPACK_COLUMN**.

XmNorientation
: This resource determines whether **RowColumn** layouts are row-major or column-major. In a column-major layout, the children of the **RowColumn** are laid out in columns top to bottom within the widget. In a row-major layout the children of the **RowColumn** are laid out in rows.

XmNpacking	The value of this resource determines how the row column widget lays out its children. When a **RowColumn** widget packs the items it contains, it determines its major dimension using the value of the **XmNorientation** resource. These are the possible values:

- **XmPACK_TIGHT**: indicates that given the current major dimension (for example, vertical if **XmNorientation** is **XmVERTICAL**), entries are placed one after the other until the **RowColumn** widget must wrap. **RowColumn** wraps when there is no room left for a complete child in that dimension. Wrapping occurs by beginning a new row or column in the next available space. Wrapping continues, as often as necessary, until all of the children are laid out.
- **XmPACK_COLUMN**: indicates that all entries are placed in identically sized boxes. The box is based on the largest height and width values of all the children widgets. The value of the **XmNnumColumns** resource determines how many boxes are placed in the major dimension, before extending in the minor dimension.
- **XmPACK_NONE**: indicates that no packing is performed. The X and Y attributes of each entry are left alone, and the **RowColumn** widget attempts to become large enough to enclose all entries.

RadioBox

The **RadioBox** is really a **RowColumn** widget configured to force one-of-many behavior on its children, which must be toggle buttons. RapidApp creates a **RadioBox** with two default toggle buttons, which you can edit to suit your needs. You can also add more toggles. IRIS IM allows you to add arbitrary items to a **RadioBox**, but then issues warnings at run time. Because the “radio” behavior can be achieved only with toggles, RapidApp supports only toggle children.

RadioBox Resources

All the **RadioBox** resources are the same as for **RowColumn**, with the following addition:

XmNradioAlwaysOne
: If this resource is set to True, one child must always be selected.

PanedWindow, HorzPanedWindow

PanedWindow is a composite widget that tiles its children vertically. Children are positioned top-to-bottom in the order in which they are created. The **PanedWindow** grows to match the width of its widest child, and all other children are forced to this width. The height of the **PanedWindow** is equal to the sum of the heights of all its children, the spacing between them, and the size of the top and bottom margins.

The **HorzPanedWindow** is a Silicon Graphics extension to Motif that supports horizontal panes. This widget is otherwise identical to **PanedWindow**.

The user can also adjust the size of the panes using an optional sash positioned on the bottom of the pane that it controls.

The **PanedWindow** presents an interaction problem when used in a tool like RapidApp because it stretches its first child to cover the entire window, and you cannot drop additional widgets directly on the **PanedWindow** itself. There are several solutions to this issue:

Drop on a non-container child or class
: If any child of a **PanedWindow** is a Control or a class (neither of which are children), you can drop a new child on one of these widgets. The drop falls through to the **PanedWindow**. This suggests a work style for creating complex panes: create the collection of widgets to be placed in each pane separately, define as a class, and add the **PanedWindow** last.

Use Keep Parent Mode

You can select RapidApp's Keep Parent mode from the View menu, which maintains the currently selected widget as a parent regardless of where a new widget might be dropped. In Keep Parent mode, select the **PanedWindow** (using the Select Parent command if necessary) and then create new widgets without changing the selected parent.

Drop on the Sash

Once a **PanedWindow** widget has more than one child, you can drop new widgets onto a Sash, the small control located between widgets to add new panes.

PanedWindow, HorzPanedWindow Resources

Following are the **PanedWindow, HorzPanedWindow** resources:

XmNseparatorOn

Determines whether a separator is created between each of the panes. The default Value is True.

PanedWindow, HorzPanedWindow Constraint Resources

Following are the resources supported by any child of a **PanedWindow**:

XmNallowResize

If this resource is set to True, the child can be resized. Otherwise, the size of the child is held constant.

XmNpaneMinimum

The value of this resource specifies the minimum size of the child.

XmNpaneMaximum

The value of this resource specifies the maximum size of the child.

Frame

Frame is a very simple manager used to enclose a single child in a border drawn by the **Frame**. The **Frame** widget is most often used to enclose other containers to create a decorative effect. The **Frame** widget can also support a second child, generally a label, which is used as a title.

If you include a a title, it is generally best to add the title first. The title will be treated as a work area child, to be framed, when initially added. Select the child and change the XmNchildType resource to **XmFRAME_TITLE_CHILD**.

Frame Resources

Following is the **Frame** resource:

XmNshadowType

This resource controls the drawing style for the **Frame** widget, and can have the following values:

- **XmSHADOW_IN**: draws an inset border.
- **XmSHADOW_OUT**: draws **Frame** so that it appears outset.
- **XmSHADOW_ETCHED_IN**: draws **Frame** using a double line giving the effect of a line etched into the window.
- **XmSHADOW_ETCHED_OUT**: draws **Frame** using a double line giving the effect of a line coming out of the window.

Frame Constraint Resources

Following are the **Frame** constraint resources:

XmNchildType Specifies whether a child is a title or work area. **Frame** supports a single title and/or work area child. The possible values are:

- **XmFRAME_TITLE_CHILD**
- XmFRAME_WORKAREA_CHILD
- **XmFRAME_GENERIC_CHILD**

The **Frame** geometry manager ignores any child of type **XmFRAME_GENERIC_CHILD**.

XmNchildHorizontalAlignment

Specifies the alignment of the title. This resource has the following values:

- **XmALIGNMENT_BEGINNING**
- **XmALIGNMENT_CENTER**
- **XmALIGNMENT_END**

XmNchildVerticalAlignment

Specifies the vertical alignment of the title text, or the title area in relation to the top shadow of the **Frame**.

- **XmALIGNMENT_BASELINE_BOTTOM**: the baseline of the title aligns vertically with the top shadow of the **Frame**. In the case of a multiline title, the baseline of the last line of text aligns vertically with the top shadow of the **Frame**.
- **XmALIGNMENT_BASELINE_TOP**: the baseline of the first line of the title aligns vertically with the top shadow of the **Frame**.
- **XmALIGNMENT_WIDGET_TOP**: the top edge of the title area aligns vertically with the top shadow of the **Frame**.
- **XmALIGNMENT_CENTER**: the center of the title area aligns vertically with the top shadow of the **Frame**.
- **XmALIGNMENT_WIDGET_BOTTOM**: the bottom edge of the title area aligns vertically with the top shadow of the **Frame**.

ScrolledWindow

The **ScrolledWindow** widget combines one or two **ScrollBar** widgets and a viewing area to implement a visible window onto aother (usually larger) data display. The visible part of the window can be scrolled through the larger display by the use of **ScrollBar**s.

ScrolledWindow can be configured to operate automatically so that it performs all scrolling and display actions with no need for application program involvement. It can also be configured to provide a minimal support framework in which the application is responsible for processing all user input and making all visual changes to the displayed data in response to that input.

ScrolledWindow Resources

Following are the resources supported by the **ScrolledWindow** widget:

XmNscrollBarDisplayPolicy
: Controls the automatic placement of the **ScrollBar**s. If this resource is set to **XmAS_NEEDED** and if **XmNscrollingPolicy** is set to **XmAUTOMATIC**, **ScrollBar**s are displayed only if the workspace exceeds the clip area in one or both dimensions. A resource value of **XmSTATIC** causes the **ScrolledWindow** to display the **ScrollBar**s whenever they are managed, regardless of the relationship between the clip window and the work area. This resource must be **XmSTATIC** when **XmNscrollingPolicy** is **XmAPPLICATION_DEFINED**.

XmNscrollingPolicy
: Performs automatic scrolling of the work area with no application interaction. If the value of this resource is **XmAUTOMATIC**, **ScrolledWindow** automatically creates the **ScrollBar**s, attaches callbacks to the **ScrollBar**s, and automatically moves the work area through the clip window in response to any user interaction with the **ScrollBar**s.

  When **XmNscrollingPolicy** is set to **XmAPPLICATION_DEFINED**, the application is responsible for all aspects of scrolling. The **ScrollBar**s must be created by the application, and it is responsible for performing any visual changes in the work area in response to user input.

RubberBoard

The **RubberBoard** widget employs a novel layout algorithm that relies on you teaching the widget how its children should be positioned, as well as how they should behave when the **RubberBoard** is resized. Using the **RubberBoard** requires the following simple steps, which must be performed exactly in sequence:

1. Make the **RubberBoard** as small as it could ever reasonably be.
2. Position all children as they would be positioned and sized for the current **RubberBoard** size.
3. Select the **RubberBoard** and set the **XmNsetInitial** resource to True, to take a "snapshot" of the current layout.
4. Resize the **RubberBoard** to its largest reasonable size.
5. Lay out the children again and resize them as you would expect them to appear for the current **RubberBoard** size.
6. Select the **RubberBoard** and set the **XmNsetFinal** resource to True.

From this point, the children will resize and reposition based on an interpolation of the two layouts you have provided.

RubberBoard Resources

Following are the resources supported by the **RubberBoard** widget:

XmNsetFinal	Switching this resource to True forces the widget to record the final positions and sizes of all children.
XmNsetInitial	Switching this resource to True forces the widget to record the initial positions and sizes of all children.

DrawingArea, VisualDrawingArea

DrawingArea is an empty widget that invokes callbacks to notify the application when graphics need to be drawn (exposure events or widget resize) and when the widget receives input from the keyboard or mouse.

Applications are responsible for defining appearance and behavior as needed in response to **DrawingArea** callbacks. The **DrawingArea** widget is typically used to display graphics drawn using Xlib functions.

The **VisualDrawingArea** is a Silicon Graphics extension that differs from the normal Motif **DrawingArea** in its support for Visual types.

DrawingArea, VisualDrawingArea Resources

Following are the resources supported by both the **DrawingArea** and **VisualDrawingArea** widgets:

XmNexposeCallback
Specifies the member function to be called when **DrawingArea** receives an exposure event. The callback reason is **XmCR_EXPOSE**. The callback structure also includes the exposure event.

The default bit gravity for this widget is **NorthWestGravity**, which may cause the **XmNexposeCallback** not to be invoked when the **DrawingArea** window is made smaller.

XmNinputCallback
Specifies the member function to be called when the **DrawingArea** receives a keyboard or mouse event (key or button, up or down). The callback reason is **XmCR_INPUT**. The callback structure also includes the input event.

XmNresizeCallback
Specifies the member function to be called when the **DrawingArea** is resized. The callback reason is **XmCR_RESIZE**.

VisualDrawingArea Resources

Following are the resources supported by the **VisualDrawingArea** widget only:

SgNditherBackground
if this resource is true and if the visual used with this widget is a TrueColor or StaticColor visual, and the widget is unable to get an exact match for the requested background color, the widget attempts to produce a dithered pixmap that produces a closer background to that requested. If one is found, it will automatically set the **XmNbackgroundPixmap** resource to this pixmap. See the **SgVisualDrawingArea** reference page for more details.

SgNinstallColormap

If this resource is set to True, this resource specifies that the widget should set the **WM_COLORMAP_WINDOWS** property on th4e shell that contains this widget, so the window manager will install the colomap when the application gets focus. See the **SgVisualDrawingArea** reference page for more details.

Controls Palette

The Controls palette (see Figure B-12) contains controls interface elements such as text field, finder, and scroll bar.

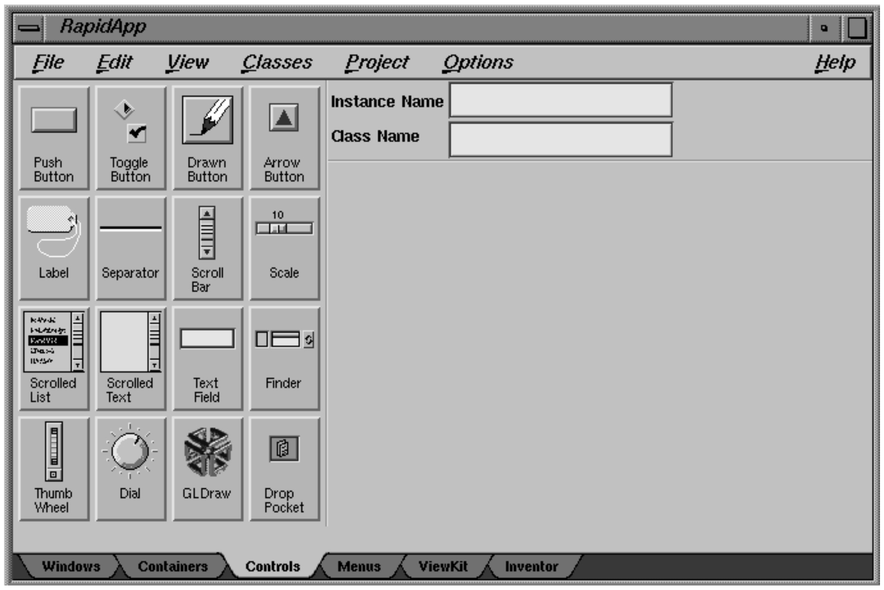


Figure B-12 Controls Palette

The user interface elements available through this palette are described in the following sections.

PushButton

The **PushButton** widget issues commands within an application. It consists of a text label or pixmap surrounded by a border shadow. When a **PushButton** is selected, the shadow changes to give the appearance that it has been pressed in. When a **PushButton** is unselected, the shadow changes to give the appearance that it is out.

PushButton Resources

Following are the resources supported by the **PushButton** widget:

XmNactivateCallback

Specifies the list of callbacks that is called when **PushButton** is activated. **PushButton** is activated when the user presses and releases the active mouse button while the pointer is inside that widget. Activating the **PushButton** also disarms it. For this callback the reason is **XmCR_ACTIVATE**.

XmNalignment

Specifies the label alignment for text or pixmap.

- **XmALIGNMENT_BEGINNING** (left alignment): the left sides of the lines of text are vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
- **XmALIGNMENT_CENTER** (center alignment): the centers of the lines of text are vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window.
- **XmALIGNMENT_END** (right alignment): the right sides of the lines of text are vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

XmNlabelPixmap

Specifies the pixmap when XmNlabelType is **XmPIXMAP**. The default value, **XmUNSPECIFIED_PIXMAP**, displays an empty label. Setting this resource in RapidApp automatically sets the **XmNlabelType** to **XmPIXMAP**. In RapidApp, pixmaps are specified as a filename. The file may be a XPM pixmap or an X bitmap. If the pixmap is loaded successfully, its base name is extracted and used as the name of the pixmap. The pixmap is always written out to a file, *pixmaps.h* in generated code, as an XPM pixmap.

Besides typing in the name of a file, you can also drop a file into the drop pocket beside the input field, or drop a pixmap file directly on the widget whose pixmap is to be set.

XmNlabelString
Specifies the string to be displayed when the XmNlabelType is **XmSTRING**. In RapidApp, setting or changing this resource automatically sets the value of **XmNlabelType** to **XmSTRING**.

XmNlabelType Specifies the label type.

- **XmSTRING**: displays text using **XmNlabelString**.
- **XmPIXMAP**: displays pixmap using **XmNlabelPixmap** or **XmNlabelInsensitivePixmap**.

Changing either the **XmNlabelString** or **XmNlabelPixmap** in RapidApp automatically sets the resource.

XmNrecomputeSize
Specifies a Boolean value that indicates whether the widget shrinks or expands to accommodate its contents (label string or pixmap) as a result of an **XtSetValues** resource value that would change the size of the widget. If this resource is set to True, the widget shrinks or expands to exactly fit the label string or pixmap. If this resource is set to False, the widget never attempts to change size on its own.

Code Examples

Programs most often use the **PushButton** widget as an input device and simply respond to a callback when the button is pushed. This is a typical member function created by RapidApp for handling a **PushButton**:

```
AClass::handlePushButton(Widget w, XtPointer callData )
{
  XmAnyCallbakStruct *cbs = (XmAnyCallbackStruct*) callData;

  //--- Comment out the following line when
  // AClass::handlePushButton is implemented

  ::VkUnimplemented ( w, "AClass::handlePushButton");

  // Add application code for AClass::handlePushButton here:

}
```

The first line makes the callData passed by all IRIS IM callbacks available in its generic form. For **PushButton** widgets, you may wish to change the cast to **XmPushButtonCallbackStruct**. The `::VkUnimplemented` call is useful when using the Developer Magic debugger and for printing a trace of this callback. You can comment it out once it is no longer needed.

A **PushButton** is a subclass of the Label widget, so the appearance of the **PushButton** can be manipulated the same as the Label widget. For example, consider the following code:

Example B-1 Retrieving Text from a Subclass of Label Using the IRIS IM API

```
XmString xmstr;
char *text;
XtVaGetValues( widget, XmNlabelString, &xmstr, NULL);
text = XmStringGetLtoR(xmstr, XmFONTLIST_DEFAULT_TAB);
```

Example B-2 Retrieving Text from a Subclass of Label Using the VkEZ API

```
char *text = EZ(widget);
Setting text on a Subclass of Label using the IRIS IM API
XmString xmstr;
xmstr = XmStringCreateLtoR("text", XmFONTLIST_DEFAULT_TAG);
XtVaSetValues(widget, XmNlabelString, xmstr, NULL);
```

The following also works:

```
XtVaSetValues(widget, XtVaTypedArg, XmNlabelString,
              XmRString, "text", strlen("text") + 1, NULL);
```

Example B-3 Setting Text on a Subclass of Label using the VkEZ API

```
EZ(widget) = "text";
```

ToggleButton

ToggleButton is used to toggle between two states. Usually this widget consists of an indicator (square or diamond) with either text or a pixmap on one side of it. However, it can also consist of just text or a pixmap without the indicator.

The toggle graphics display a 1-of-many or N-of-many selection state. When a toggle indicator is displayed, a square indicator shows an N-of-many selection state and a diamond indicator shows a 1-of-many selection state.

ToggleButton Resources

Following are the **ToggleButton** resources:

XmNalignment

Specifies the label alignment for text or pixmap.

- **XmALIGNMENT_BEGINNING** (left alignment): the left sides of the lines of text are vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
- **XmALIGNMENT_CENTER** (center alignment): the centers of the lines of text are vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window.
- **XmALIGNMENT_END** (right alignment): the right sides of the lines of text are vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

XmNindicatorOn

Specifies that a toggle indicator is drawn to one side of the toggle text or pixmap when set to True. When set to False, no space is allocated for the indicator, and it is not displayed.

XmNlabelPixmap

Specifies the pixmap when **XmNlabelType** is **XmPIXMAP**. The default value, **XmUNSPECIFIED_PIXMAP**, displays an empty label. Setting this resource in RapidApp automatically sets the **XmNlabelType** to **XmPIXMAP**. In RapidApp, pixmaps are specified as a filename. The file may be an XPM pixmap or an X bitmap. If the pixmap is loaded successfully, its base name is extracted and used as the name of the pixmap. The pixmap is always written out to a file, *pixmaps.h* in generated code, as an XPM pixmap.

Besides typing in the name of a file, you can also drop a file into the drop pocket beside the input field, or drop a pixmap file directly on the widget whose pixmap is to be set.

XmNlabelString

Specifies the string to be displayed when the **XmNlabelType** is **XmSTRING**. In RapidApp, setting or changing this resource automatically sets the value of **XmNlabelType** to **XmSTRING**.

XmNlabelType

Specifies the label type.

- **XmSTRING**: displays text using **XmNlabelString**.
- **XmPIXMAP**: displays pixmap using **XmNlabelPixmap** or XmNlabelInsensitivePixmap.

Changing either the **XmNlabelString** or **XmNlabelPixmap** in RapidApp automatically sets the resource.

XmNrecomputeSize

Specifies a Boolean value that indicates whether the widget shrinks or expands to accommodate its contents (label string or pixmap) as a result of an **XtSetValues** resource value that would change the size of the widget. If this resource is set to True, the widget shrinks or expands to exactly fit the label string or pixmap. If this resource is set to False, the widget never attempts to change size on its own.

XmNselectPixmap
Specifies the pixmap to be used as the button face if **XmNlabelType** is **XmPIXMAP** and the **ToggleButton** is selected. When the **ToggleButton** is unselected, the pixmap specified in Label's **XmNlabelPixmap** is used. If no value is specified for **XmNlabelPixmap**, that resource is set to the value specified for **XmNselectPixmap**.

XmNset Represents the state of the **ToggleButton**. A value of false indicates that the **ToggleButton** is not set. A value of true indicates that the **ToggleButton** is set. Setting this resource sets the state of the **ToggleButton**.

XmNvalueChangedCallback
Specifies the list of callbacks called when the **ToggleButton** value is changed. To change the value, press and release the active mouse button while the pointer is inside the **ToggleButton**. This action also causes this widget to be disarmed. For this callback, the reason is **XmCR_VALUE_CHANGED**.

Code Examples

Following are examples of **ToggleButton** use:

Example B-4 Setting the Indicator State on a Toggle Button Without Invoking Callbacks

```
XtVaSetValues(widget, XmNset, newBooleanValue, NULL);
```

Example B-5 Setting the Indicator State on a Toggle Button and Triggering Callbacks

```
XmToggleButtonSetState(widget, newBooleanValue, True);
```

ArrowButton

The arrow button widget is similar to the PushButton widget, but is displayed as a directional arrow.

Resources

Following are **ArrowButton** resources:

XmNarrowDirection
Determines the arrow direction.

XmNactivateCallback
The member function to be called when the arrow button is pressed.

DrawnButton

The **DrawnButton** widget consists of an empty widget window surrounded by a shadow border. It provides the application developer with a graphics area that can have **PushButton** input semantics.

Callback types are defined for widget exposure and widget resize to allow the application to redraw or reposition its graphics.

DrawnButton Resources

Following are the **DrawnButton** resources:

XmNactivateCallback
Specifies the list of callbacks that is called when the **DrawnButton** is activated. **DrawnButton** is activated when the user presses and releases the active mouse button while the pointer is inside that widget. Activating the **DrawnButton** also disarms it. For this callback, the reason is **XmCR_ACTIVATE**.

XmNalignment
Specifies the label alignment for text or pixmap.

- **XmALIGNMENT_BEGINNING** (left alignment): the left sides of the lines of text are vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.

- **XmALIGNMENT_CENTER** (center alignment): the centers of the lines of text are vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window.
- **XmALIGNMENT_END** (right alignment): the right sides of the lines of text are vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

XmNexposeCallback

Specifies the member function to be called when **DrawnButton** needs to be redrawn.

Specifies the list of callbacks that is called when the widget receives an exposure event. The reason sent by the callback is **XmCR_EXPOSE**.

XmNlabelPixmap

Specifies the pixmap when **XmNlabelType** is **XmPIXMAP**. The default value, **XmUNSPECIFIED_PIXMAP**, displays an empty label. Setting this resource in RapidApp automatically sets the **XmNlabelType** to **XmPIXMAP**. In RapidApp, pixmaps are specified as a filename. The file may be an XPM pixmap or an X bitmap. If the pixmap is loaded successfully, its base name is extracted and used as the name of the pixmap. The pixmap is always written out to a file, *pixmaps.h* in generated code, as an XPM pixmap.

Besides typing in the name of a file, you can also drop a file into the drop pocket beside the input field, or drop a pixmap file directly on the widget whose pixmap is to be set.

XmNlabelString

Specifies the string to be displayed when the **XmNlabelType** is **XmSTRING**. In RapidApp, setting or changing this resource automatically sets the value of **XmNlabelType** to **XmSTRING**.

XmNlabelType Specifies the label type.

- **XmSTRING: displays text using XmNlabelString**.
- **XmPIXMAP**: displays pixmap using **XmNlabelPixmap** or **XmNlabelInsensitivePixmap**.

Changing either the **XmNlabelString** or **XmNlabelPixmap** in RapidApp automatically sets the resource.

XmNpushButtonEnabled
Enables or disables the three-dimensional shadow drawing as in **PushButton**.

XmNrecomputeSize
Specifies a Boolean value that indicates whether the widget shrinks or expands to accommodate its contents (label string or pixmap) as a result of an **XtSetValues** resource value that would change the size of the widget. If this resource is set to True, the widget shrinks or expands to exactly fit the label string or pixmap. If this resource is set to False, the widget never attempts to change size on its own.

Label

The **Label** widget can contain non-editable text or a pixmap.

Label Resources

The **Label** widget supports the following resources:

XmNalignment Specifies the label alignment for text or pixmap.

- **XmALIGNMENT_BEGINNING** (left alignment): the left sides of the lines of text are vertically aligned with the left edge of the widget window. For a pixmap, its left side is vertically aligned with the left edge of the widget window.
- **XmALIGNMENT_CENTER** (center alignment): the centers of the lines of text are vertically aligned in the center of the widget window. For a pixmap, its center is vertically aligned with the center of the widget window.

- **XmALIGNMENT_END** (right alignment): the right sides of the lines of text are vertically aligned with the right edge of the widget window. For a pixmap, its right side is vertically aligned with the right edge of the widget window.

XmNlabelPixmap
Specifies the pixmap when **XmNlabelType** is **XmPIXMAP**. The default value, **XmUNSPECIFIED_PIXMAP**, displays an empty label. Setting this resource in RapidApp automatically sets the **XmNlabelType** to **XmPIXMAP**. In RapidApp, pixmaps are specified as a filename. The file may be a XPM pixmap or an X bitmap. If the pixmap is loaded successfully, its base name is extracted and used as the name of the pixmap. The pixmap is always written out to a file, *pixmaps.h* in generated code, as an XPM pixmap.

Besides typing in the name of a file, you can also drop a file into the drop pocket beside the input field, or drop a pixmap file directly on the widget whose pixmap is to be set.

XmNlabelString
Specifies the string to be displayed when the **XmNlabelType** is **XmSTRING**. In RapidApp, setting or changing this resource automatically sets the value of **XmNlabelType** to **XmSTRING**.

XmNlabelType Specifies the label type.

- **XmSTRING**: displays text using **XmNlabelString**.
- **XmPIXMAP**: displays pixmap using **XmNlabelPixmap** or **XmNlabelInsensitivePixmap**.

Changing either the **XmNlabelString** or **XmNlabelPixmap** in RapidApp automatically sets the resource.

XmNrecomputeSize
Specifies a Boolean value that indicates whether the widget shrinks or expands to accommodate its contents (label string or pixmap) as a result of an **XtSetValues** resource value that would change the size of the widget. If this

resource is set to True, the widget shrinks or expands to exactly fit the label string or pixmap. If this resource is set to False, the widget never attempts to change size on its own.

Code Examples

Following are examples of **Label** use:

Example B-6 Retrieving Text from a Subclass of Label Using the IRIS IM API

```
XmString xmstr;
char *text;
XtVaGetValues( widget, XmNlabelString, &xmstr, NULL);
text = XmStringGetLtoR(xmstr, XmFONTLIST_DEFAULT_TAB);
```

Example B-7 Retrieving Text from a Subclass of Label Using the VkEZ API

```
char *text = EZ(widget);
Setting text on a Subclass of Label using the IRIS IM API
XmString xmstr;
xmstr = XmStringCreateLtoR("text", XmFONTLIST_DEFAULT_TAG);
XtVaSetValues(widget, XmNlabelString, xmstr, NULL);
```

The following is also valid:

```
XtVaSetValues(widget, XtVaTypedArg, XmNlabelString,
              XmRString, "text", strlen("text") + 1, NULL);
```

Example B-8 Setting Text on a Subclass of Label Using the VkEZ API

```
EZ(widget) = "text";
```

Separator

Separator is a primitive widget that separates items in a display. Several different line drawing styles are provided, as well as horizontal or vertical orientation.

The **Separator** line drawing is automatically centered within the height of the widget for a horizontal orientation and centered within the width of the widget for a vertical orientation.

Separator Resources

The **Separator** widget supports the following resources:

XmNorientation
Displays **Separator** vertically or horizontally. This resource can have values of **XmVERTICAL** and **XmHORIZONTAL**.

XmNseparatorType
Specifies the type of line drawing to be done in the **Separator** widget.

- **XmSINGLE_LINE**: single line.
- **XmDOUBLE_LINE**: double line.
- **XmSINGLE_DASHED_LINE**: single-dashed line.
- **XmDOUBLE_DASHED_LINE**: double-dashed line.
- **XmNO_LINE**: no line.
- **XmSHADOW_ETCHED_IN**: a line whose shadows give the effect of a line etched into the window.
- **XmSHADOW_ETCHED_OUT**: a line whose shadows give the effect of an etched line coming out of the window.
- **XmSHADOW_ETCHED_IN_DASH**: identical to **XmSHADOW_ETCHED_IN** except a series of lines creates a dashed line.
- **XmSHADOW_ETCHED_OUT_DASH**: identical to **XmSHADOW_ETCHED_OUT** except a series of lines creates a dashed line.

ScrollBar

The **ScrollBar** widget allows the user to view data that is too large to be displayed all at once. **ScrollBar**s are usually located inside a **ScrolledWindow** and adjacent to the widget that contains the data to be viewed. When the user interacts with the **ScrollBar**, the data within the other widget scrolls.

A **ScrollBar** consists of two arrows placed at each end of a rectangle. The rectangle is called the scroll region. A smaller rectangle, called the slider, is placed within the scroll region. The data is scrolled by clicking either arrow, clicking the scroll region, or dragging the slider. When an arrow is selected, the slider within the scroll region is moved in the direction of the arrow by an amount supplied by the application. If the mouse button is held down, the slider continues to move at a constant rate.

ScrollBar Resources

The following resources are available for the **ScrollBar** widget from within RapidApp:

XmNdragCallback
Specifies the list of callbacks that is called on each incremental change of position when the slider is being dragged. The reason sent by the callback is **XmCR_DRAG**.

XmNorientation
Specifies whether the **ScrollBar** is displayed vertically or horizontally. This resource can have values of **XmVERTICAL** and **XmHORIZONTAL**.

XmNvalueChangedCallback
Specifies the list of callbacks that is called when the slider is released after being dragged. The reason passed to the callback is **XmCR_VALUE_CHANGED**.

Code Examples

Following are examples of **ScrollBar** use:

Example B-9 Getting the Value of a Scroll Bar Using the IRIS IM API

```
int value;
XtVaGetValues(widget, XmNvalue, &value, NULL);
```

Example B-10 Getting the Value of a Scroll Bar Using the VkEZ API

```
int value = EZ(widget);
```

Example B-11 Setting the Value of a Scroll Bar Using the IRIS IM API

```
XtVaSetValues(widget, XmNvalue, 100, NULL);
```

Example B-12 Setting the Value of a Scroll Bar Using the VkEZ API

```
EZ(widget) = 100;
```

Scale

Scale is used by an application to indicate a value from within a range of values, and it allows the user to input or modify a value from the same range.

A **Scale** has an elongated rectangular region similar to a **ScrollBar**. A slider inside this region indicates the current value along the **Scale**. The user can also modify the **Scale**'s value by moving the slider within the rectangular region of the **Scale**. A **Scale** can also include a label set located outside the **Scale** region. These can indicate the relative value at various positions along the scale.

A **Scale** can be either input / output or output only. An input / output **Scale**'s value can be set by the application and also modified by the user with the slider. An output-only **Scale** is used strictly as an indicator of the current value of something and cannot be modified interactively by the user.

Scale Resources

The **Scale** widget supports the following resources:

XmNdecimalPoints
Specifies the number of decimal points to shift the slider value when displaying it. For example, a slider value of 2,350 and an **XmNdecimalPoints** value of 2 results in a display value of 23.50. The value must not be negative.

XmNdragCallback
Specifies the list of callbacks that is called when the slider position changes as the slider is being dragged. The reason sent by the callback is **XmCR_DRAG**.

XmNmaximum Specifies the slider's maximum value. **XmNmaximum** must be greater than **XmNminimum**.

XmNminimum Specifies the slider's minimum value. **XmNmaximum** must be greater than **XmNminimum**.

XmNorientation
Displays **Scale** vertically or horizontally. This resource can have values of **XmVERTICAL** and **XmHORIZONTAL**.

XmNscaleHeight
Specifies the height of the slider area. The value should be in the specified unit type (the default is pixels). If no value is specified, a default height is computed.

XmNscaleWidth
Specifies the width of the slider area. The value should be in the specified unit type (the default is pixels). If no value is specified, a default width is computed.

XmNshowValue
Specifies whether a label for the current slider value should be displayed next to the slider. If the value is True, the current slider value is displayed.

XmNtitleString
Specifies the title text string to appear in the **Scale** widget window.

XmNvalue Specifies the slider's current position along the scale, between **XmNminimum** and **XmNmaximum**. The value must be within these inclusive bounds. The initial value of this resource is the larger of 0 and **XmNminimum**.

XmNvalueChangedCallback
Specifies the list of callbacks that is called when the value of the slider has changed. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

Code Examples

Following are examples of **Scale** use:

Example B-13 Getting the Value of a Scale Using the IRIS IM API

```
int value;
XtVaGetValues(widget, XmNvalue, &value, NULL);
```

Example B-14 Getting the Value of a Scale Using the VkEZ API

```
int value = EZ(widget);
```

Example B-15 Setting the Value of a Scale Using the IRIS IM API

```
XtVaSetValues(widget, XmNvalue, 100, NULL);
```

Example B-16 Setting the Value of a Scale Using the VkEZ API

```
EZ(widget) = 100;
```

ScrolledList

ScrolledList allows a user to select one or more items from a group of choices. Items are selected from the list in a variety of ways, using both the pointer and the keyboard. **ScrolledList** operates on an array of compound strings that are defined by the application. Each compound string becomes an item in the **ScrolledList**, with the first compound string becoming the item in position 1, the second becoming the item in position 2, and so on.

Each list has one of four selection models:

- Single Select
- Browse Select
- Multiple Select
- Extended Select

In Single Select and Browse Select, only one item is selected at a time. In Single Select, pressing BSelect on an item toggles its selection state and deselects any other selected item. In Browse Select, pressing BSelect on an item selects it and deselects any other selected item; dragging BSelect moves the selection as the pointer is moved. Releasing BSelect on an item moves the location cursor to that item.

In Multiple Select, any number of items can be selected at a time. Pressing BSelect on an item toggles its selection state but does not deselect any other selected items.

In Extended Select, any number of items can be selected at a time, and the user can easily select ranges of items. Pressing BSelect on an item selects it and deselects any other selected item. Dragging BSelect or pressing or dragging BExtend following a BSelect action selects all items between the item under the pointer and the item on which BSelect was pressed. This action also deselects any other selected items outside that range.

Scrolled Window Resources

The following resources are supported by the **ScrolledWindow** that contains the List widget. You can select the **ScrolledWindow** by clicking on the **ScrollBar** area, or using the “Select Parent” command.

XmNscrollBarDisplayPolicy
Controls the automatic placement of the **ScrollBar**s. If this resource is set to **XmAS_NEEDED** and if **XmNscrollingPolicy** is set to **XmAUTOMATIC**, **ScrollBar**s are displayed only if the workspace exceeds the clip area in one or both dimensions. A resource value of **XmSTATIC** causes the **ScrolledWindow** to display the **ScrollBar**s whenever they are managed, regardless of the relationship between the clip window and the work area. This resource must be **XmSTATIC** when **XmNscrollingPolicy** is **XmAPPLICATION_DEFINED**.

XmNscrollingPolicy
Performs automatic scrolling of the work area with no application interaction. If the value of this resource is **XmAUTOMATIC**, **ScrolledWindow** automatically creates the **ScrollBar**s, attaches callbacks to the **ScrollBar**s, and automatically moves the work area through the clip window in response to any user interaction with the **ScrollBar**s.

When **XmNscrollingPolicy** is set to **XmAPPLICATION_DEFINED**, the application is responsible for all aspects of scrolling. The **ScrollBar**s must be created by the application, and it is responsible for performing any visual changes in the work area in response to user input.

List Resources

The following resources are supported by the **List** widget. Click in the list area to access these resources.

XmNbrowseSelectionCallback
Specifies the member function to be called when an item is selected in the browse selection mode. The reason is **XmCR_BROWSE_SELECT**.

XmNdefaultActionCallback
Specifies the member function to be called when an item is double-clicked or KActivate is pressed. The reason is **XmCR_DEFAULT_ACTION**.

XmNextendedSelectionCallback
Specifies the member function to be called when items are selected using the extended selection mode.

XmNitems Points to an array of compound strings that are to be displayed as the list items. In RapidApp, static or initial items can be entered as a comma-separated list.

XmNlistSizePolicy
Controls the reaction of the **List** when an item grows horizontally beyond the current size of the **List** work area. If the value is **XmCONSTANT**, the list viewing area does not grow, and a horizontal **ScrollBar** is added for a **ScrolledList**. If this resource is set to **XmVARIABLE**, the **List** grows to match the size of the longest item, and no horizontal **ScrollBar** appears.

When the value of this resource is **XmRESIZE_IF_POSSIBLE**, the **List** attempts to grow or shrink to match the width of the widest item. If it cannot grow to match the widest size, a horizontal **ScrollBar** is added for a **ScrolledList** if the longest item is wider than the list viewing area.

XmNmultipleSelectionCallback
Specifies the member function to be called when an item is selected in multiple selection mode.

XmNselectionPolicy
Defines the interpretation of the selection action. This can be one of the following:

- **XmSINGLE_SELECT**: allows only single selections
- **XmMULTIPLE_SELECT**: allows multiple selections
- **XmEXTENDED_SELECT**: allows extended selections
- **XmBROWSE_SELECT**: allows "drag and browse" functionality

XmNsingleSelectionCallback
Specifies the member function to be called when an item is selected in single selection mode.

XmNvisibleItemCount
Specifies the number of items that can fit in the visible space of the list work area. The **List** uses this value to determine its height. The value must be greater than 0.

Scrolled Text

The Scrolled Text widget provides a simple multi-line scrollable text editor.

Scrolled Text Resources

Following are the resources supported by the **ScrolledText** widget:

XmNcolumns Determines the width of the widget in terms of the number of characters that can be displayed horizontally.

XmNeditable Indicates that the user can edit the text string when set to True. Prohibits the user from editing the text when set to False. In RapidApp and RapidApp-generated code, the Text widget automatically changes to read-only color when **XmNeditable** is set to False, in conformance with the *Indigo Magic User Interface Guidelines*.

XmNmodifyVerifyCallback
Specifies the member function to be called before text is deleted from or inserted into **Text**. The type of the structure whose address is passed to this callback is **XmTextVerifyCallbackStruct**. The reason sent by the callback is **XmCR_MODIFYING_TEXT_VALUE**.

XmNmotionVerifyCallback
Specifies the member function to be called before the insert cursor is moved to a new position. The type of the structure whose address is passed to this callback is **XmTextVerifyCallbackStruct**. The reason sent by the callback is **XmCR_MOVING_INSERT_CURSOR**. It is possible for more than one **XmNmotionVerifyCallback** to be generated from a single action.

XmNrows Specifies the initial height of the text window measured in character heights. The value must be greater than 0. The default value depends on the value of the **XmNheight** resource. If no height is specified, the default is 1.

XmNscrollHorizontal
Adds a **ScrollBar** that allows the user to scroll horizontally through text when the Boolean value is True. This resource is forced to False when the **Text** widget is placed in a **ScrolledWindow** with **XmNscrollingPolicy** set to **XmAUTOMATIC**.

XmNvalue Specifies the initial contents of the **Text** widget.

XmNvalueChangedCallback
Specifies the member function to be called after text is deleted from or inserted into **Text**. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

TextField

TextField is a simple, single line text editor. It is similar to the **ScrolledText** widget, but can have only a single row of text and is not scrollable.

TextField Resources

Following are the resources supported by the **TextField** widget:

XmNactivateCallback
Specifies the member function to be called when the user presses **`<Enter>`**. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason sent by the callback is **XmCR_ACTIVATE**.

XmNcolumns Determines the width of the widget in terms of the number of characters that can be displayed horizontally.

XmNeditable Indicates that the user can edit the text string when set to True. Prohibits the user from editing the text when set to False. In RapidApp and RapidApp-generated code, the **Text** widget automatically changes to read-only color when **XmNeditable** is set to False, in conformance with the *Indigo Magic User Interface Guidelines*.

XmNmodifyVerifyCallback
Specifies the member function to be called before text is deleted from or inserted into **Text**. The type of the structure whose address is passed to this callback is **XmTextVerifyCallbackStruct**. The reason sent by the callback is **XmCR_MODIFYING_TEXT_VALUE**.

XmNmotionVerifyCallback
Specifies the member function to be called before the insert cursor is moved to a new position. The type of the structure whose address is passed to this callback is **XmTextVerifyCallbackStruct**. The reason sent by the callback is **XmCR_MOVING_INSERT_CURSOR**. It is possible for more than one **XmNmotionVerifyCallbacks** to be generated from a single action.

XmNvalue Specifies the initial contents of the **Text** widget.

XmNvalueChangedCallback

Specifies the member function to be called after text is deleted from or inserted into **Text**. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

Finder

The **Finder** widget integrates a **DropPocket** pocket, a **TextField**, a **ZoomBar**, and a history menu into a single widget. The **zoomBar** is a set of buttons above the text field that allows sections of the text to be selected. The history menu allows users to select items previously visited, or to undo operations. The **Finder** widget should be used for accelerating text selection of long objects such as filenames.

Clicking the *History* button brings up a pulldown menu. Selecting an item from the menu sets the text field to that item. Whenever the text field is set, the **zoomBar** changes to reflect the text sections in the text field.

Pressing a button on the **zoomBar** sets the text field to the portion of the text preceding that button. The specific behavior is customizable, but generally cuts off the portion of the text after the pressed **zoomBar** button. The history menu can be used to go back to the original text.

The **Finder** also includes a **DropPocket** for displaying icons representing entries in the **Finder**'s text field. These icons are Silicon Graphic's environment file icons. File icons from FrameMaker, Searchbook, or similar applications can be dropped on the **DropPocket**.

Finder Resources

Following are the resources supported by the **Finder** widget:

XmNactivateCallback

This callback is called when a **zoomBar** button is pushed or when the text field generates an **activateCallback** (in other words, pressing `<Enter>` in the text field) or if the text field is set by **SgFinderSetTextString**. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason sent by the callback is **XmCR_ACTIVATE**.

XmNvalueChangedCallback
The value changed callback specifies the list of callbacks that is called after text is deleted from or inserted into the text field. The type of the structure whose address is passed to this callback is **XmAnyCallbackStruct**. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

Thumbwheel

ThumbWheel is used by an application to allow the user to input or modify a value either from within a range of values or from an unbounded (infinite) range.

A **ThumbWheel** has an elongated rectangular region within which a wheel graphic is displayed. The user can modify the **ThumbWheel**'s value by spinning the wheel. A **ThumbWheel** can also include a *Home* button located outside the wheel region. This button allows the user to set the **ThumbWheel**'s value to a known position.

Thumbwheel Resources

Following are the resources supported by the **Thumbwheel** widget:

SgNhomePosition
Specifies the known value to which the thumb wheel's value is set when the *Home* button is clicked.

XmNmaximum Specifies the thumb wheel's maximum value. **XmNmaximum** must be greater than or equal to **XmNminimum**. Setting **XmNmaximum** equal to **XmNminimum** indicates an infinite range.

XmNminimum Specifies the thumb wheel's minimum value. **XmNmaximum** must be greater than or equal to **XmNminimum**. Setting **XmNmaximum** equal to **XmNminimum** indicates an infinite range.

XmNdragCallback
Specifies a member function to be called continuously as the value of the thumb wheel changes.

SgNangleRange
Specifies the angular range, in degrees, through which the thumb wheel is allowed to rotate. This, in conjunction with **XmNmaximum** and **XmNminimum**, controls the fineness or coarseness of the wheel control when it is not infinite. If this value is set to zero, the thumb wheel has an infinite range.

The default of 150 represents roughly the visible amount of the wheel. Thus clicking at one end of the wheel and dragging the mouse to the other end gives roughly the entire range from **XmNminimum** to **XmNmaximum**.

XmNorientation
Displays **ThumbWheel** vertically or horizontally. This resource can have values of **XmVERTICAL** and **XmHORIZONTAL**.

XmNvalue Specifies the current position of the thumb wheel, between **XmNminimum** and **XmNmaximum** if the thumb wheel is not infinite.

XmNvalueChangedCallback
Specifies the member function to be called when the value of the thumb wheel has changed. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

Dial

The **Dial** widget allows a user to modify a value from within a range of values. A **Dial** has a rectangular region within which a knob or pointer graphic is displayed. The user can modify the **Dial**'s value by spinning this knob or pointer.

Dial Resources

Following are the resources supported by the **Dial** widget:

SgNdialMarkers	Specifies the number of divisions around the perimeter of the dial. A "tick mark" is drawn at each division.
XmNmaximum	Specifies the dial's maximum value. **XmNmaximum** must be greater than or equal to **XmNminimum**.
XmNminimum	Specifies the dial's minimum value. **XmNmaximum** must be greater than or equal to **XmNminimum**.
SgNstartAngle	Specifies the whole number angle (0-360) where the dial starts increasing.
SgNangleRange	Specifies the angular range, in degrees, through which the dial is allowed to rotate. This, in conjunction with **XmNmaximum** and **XmNminimum**, controls the fineness or coarseness of the dial control.
SgNdialVisual	Specifies the look of the dial, either **SgKNOB** or **SgPOINTER**.
XmNdragCallback	Specifies a member function to be called when the dial position changes as the dial is being spun. The reason sent by the callback is **XmCR_DRAG**.
XmNvalue	Specifies the current position of the dial, between **XmNminimum** and **XmNmaximum**.
XmNvalueChangedCallback	Specifies a member function to be called when the value of the dial has changed. The reason sent by the callback is **XmCR_VALUE_CHANGED**.

GLwMDrawingArea

The **GLwMDrawingArea** widget creates an empty window suitable for OpenGL drawing. It provides a window with the appropriate visual- and colormaps needed for OpenGL, based on supplied parameters.

GLwMDrawingArea also provide callbacks for redraw, resize, input, and initialization.

Included in the information provided when creating a **GLwMDrawingArea** is information necessary to determine the visual. This may be provided in three ways, all of them through resources.

- A specific **visualInfo** structure may be passed in. (This **visualInfo** must have been obtained elsewhere; it is the application designer's responsibility to make sure that it is compatible with the OpenGL rendering done by the application).
- An attribute list may be provided. This attribute list is formatted identically to that used for direct open GL programming.
- Each attribute can be specified as an individual resource. This method is the simplest, and is the only method that works from resource files.

In addition to allocating the visual, the **GLwMDrawingArea** also allocates the colormap unless one is provided by the application. (If a colormap is provided, the application writeris responsible for guaranteeing compatibility between the colormap and the visual). If an application creates multiple **GLwMDrawingArea** widgets with the same visual, the same colormap is used.

GLwNexposeCallback

Specifies a member function to be called when the widget receives an exposure event. The callback reason is **GLwCR_EXPOSE**. The callback structure also includes the exposure event. You generally want the application to redraw the scene.

GLwNginitCallback

Specifies a member function to be called when the widget is first realized. Since no OpenGL operations can be done before the widget is realized, this callback can be used to perform any appropriate OpenGL initialization such as creating a context. The callback reason is **GLwCR_GINIT**.

GLwNinputCallback

Specifies a member function to be called when the widget receives a keyboard or mouse event. By default, the input callback is called on each key press and key release, on each mouse button press and release, and whenever the mouse is moved while a button is pressed. However, this can be changed by providing a different translation table. The callback structure also includes the input event. The callback reason is **GLwCR_INPUT**.

The input callback is provided as a programming convenience, as it provides a convenient way to catch all input events. However, a more modular program can often be obtained by providing specific actions and translations in the application rather than by using a single catchall callback. Use of explicit translations can also provide for more customizability.

GLwNresizeCallback
Specifies the member function to be called when the **GLwMDrawingArea** is resized. The callback reason is **GLwCR_RESIZE**.

The **GLwDrawingArea** widget requires information about the visual type to be used. This information can be passed programmatically as a visual Info structure, or the individual attributes of the visual type may be specified in RapidApp. These attributes include the following:

alphaSize	An integer value that corresponds to the GLX_RED_SIZE attribute
blueSize	An integer value that corresponds to the GLX_BLUE_SIZE attribute
doubleBuffer	A Boolean value that corresponds to the GLX_DOUBLEBUFFER attribute
greenSize	An integer value that corresponds to the **GLX_GREEN_SIZE** attribute.
level	An integer value that corresponds to the **GLX_LEVEL** attribute.
redSize	An integer value that corresponds to the **GLX_RED_SIZE** attribute.
rgba	A Boolean value that corresponds to the **GLX_RGBA** attribute.

For more information about these atributes and visual types, see the reference pages for the **GLwDrawingArea** widget, the reference page for **glxChooseVisual**, and the OpenGL specification.

Drop Pocket

The **DropPocket** widget is designed to recieve desktop icons from the IRIS Indigo Magic desktop. The **DropPocket** displays the file icon as a visual reminder of the file associated with the **DropPocket**. See the **SgDropPocket** reference page for more details.

DropPocket Resources

Following are the resources for **DropPocket**:

SgNiconUpdateCallback
The member function to be invoked when an icon is dropped in the **DropPocket**. See the **SgDropPocket** reference page for more details.

SgNname Specifies the name of the current icon. This resource can be set to specify the initial icon that appears in the **DropPocket**.

Menus Palette

The Menus palette (see Figure B-13) contains menu interface elements such as pulldown menu, option menu, and menu separator.

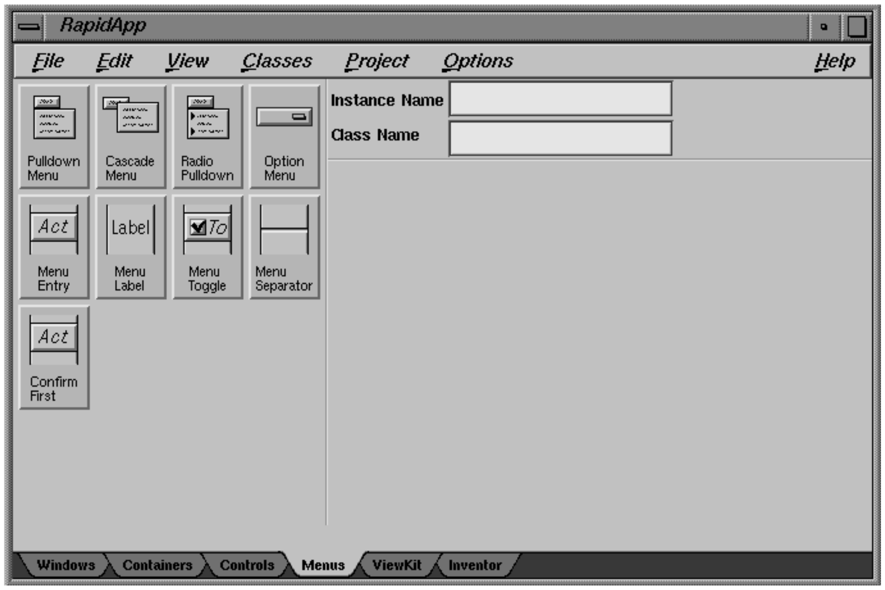


Figure B-13 Menus Palette

The user interface elements available through this palette are described in the following sections.

Pulldown Menu

The **Pulldown** menu item adds a pulldown menu to a menu bar. By default several items are included. These can be edited or removed as needed. A **Pulldown** menu is created by calling the ViewKit member function **addSubMenu()**.

You can display the menu pane by selecting it, and then clicking once again. Once displayed, you can add additional items (**MenuEntry**, **MenuLabel**, **MenuToggle**, **MenuSeparator**, **ConfirmFirst**, or other pulldowns) by droping new elements on the displayed menu area. You can dismiss the option menu by clicking on the pulldown again.

Pulldown Resources

Resources in the **Pulldown** menu correspond to the visible menu entry for this pulldown. The resources available are the following:

XmNlabelString
The string displayed for this menu pane.

XmNmnemonic
The mnemonic used to post this menu item.

Cascade Menu

The **Cascade** menu item adds a pull-right menu to an existing menu pane. By default several items are included. These can be edited or removed as needed.

You can display the menu pane by selecting it, and then clicking once again. Once displayed, you can add additional items (**MenuEntry**, **MenuLabel**, **MenuToggle**, **MenuSeparator**, **ConfirmFirst**, or other pulldowns) by droping new elements on the displayed menu area. You can dismiss the option menu by clicking on the pulldown again.

Note: For experience Motif developers: This item is identical to the **Pulldown** menu item, and is present as an aid to those less familiar with the Motif menu structure.

Cascade Resources

Resources in the **Cascade** menu correspond to the visible menu entry for this menu pane. The resources available are the following:

XmNlabelString
The string displayed for this menu pane.

XmNmnemonic
The mnemonic used to post this menu item.

Radio Pulldown

A **RadioPulldown** menu item can be aded to an existing menu bar or menu pane. It is meant to hold sets of toggle items that exhibit radio (one-of-many) behavior. By default, two initial toggles are created for each new **RadioPulldown**. These can be edited or removed as needed.

You can display the menu pane by selecting it, and then clicking once again. Once displayed, you can add additional items (**MenuEntry**, **MenuLabel**, **MenuToggle**, **MenuSeparator**, **ConfirmFirst**, or other pulldowns) by droping new elements on the displayed menu area. You can dismiss the option menu by clicking on the pulldown again.

RadioPulldown Resources

Resources in the **RadioPulldown** menu correspond to the visible menu entry for this menu pane. The resources available are the following:

XmNlabelString
The string displayed for this menu pane.

XmNmnemonic
The mnemonic used to post this menu item.

OptionMenu

The **OptionMenu** item creates a menu that can be sued to select one item from a set of choices. The **OptionMenu** is created with two initial options which can be edited or removed as needed. You can display the option menu by selecting it, and then clicking once again. Once displayed, you can add additional items (**MenuEntry** elements) by droping new elements on the displayed menu area. You can dismiss the option menu by clicking on the menu button (not the displayed menu pane).

You can display the menu pane by selecting it, and then clicking once again. Once displayed, you can add additional items (**MenuEntry**, **MenuLabel**, **MenuToggle**, **MenuSeparator**, **ConfirmFirst**, or other pulldowns) by droping new elements on the displayed menu area. You can dismiss the option menu by clicking on the pulldown again.

OptionMenu Resources

Following are the **OptionMenu** resources available through RapidApp:

XmNlabelString
: Determines the value of an optional string to be displayed to the left of the option menu as a title. If left empty,the title is not visible.

Menu Entry

The **MenuEntry** corresponds to an **XmPushButtonGadget**, and is intended to be added to a menu pane or **OptionMenu** as a selectable command entry. The **MenuEntry** is represented in the program as a ViewKit VkMenuAction object.

MenuEntry Resources

Following are the **MenuEntry** resources available through RapidApp:

XmNaccelerator
: A description of the accelerator keys that can be used to invoke this menu item.

XmNacceleratorText
: The text to be displayed in the item to remind the user of the accelerator.

XmNactivateCallback
: The member function to be invoked when this menu item is selected.

XmNlabelString
: The label to be displayed in this menu item.

XmNmnemonic
: The mnemonic that can be used to select this item.

XmNundoCallback
: The optional member function that should be called if this item is reversed using the ViewKit undo mechanism.

Menu Label

The **MenuLabel** corresponds to an XmLabelGadget, and is intended to be added to a menu pane or OptionMenu as a non-selectable entry. The **MenuLabel** will be represented in the program as a ViewKit **VkMenuLabel** object.

MenuLabel Resources

Following are the **MenuLabel** resources available through RapidApp:

XmNlabelString
: The label to be displayed in this menu item.

Menu Toggle

The **MenuToggle** corresponds to an **XmToggleButtonGadget**, and is intended to be added to a menu pane as a selectable two-state entry. The **MenuToggle** is represented in the program as a ViewKit **VkMenuToggle** object. When added to a **RadioPulldown**, this entry has one-of-many behavior. Otherwise, all toggles can be selected independently.

MenuToggle Resources

Following are the **MenuToggle** resources available through RapidApp:

XmNaccelerator
: A description of the accelerator keys that can be used to invoke this menu item.

XmNacceleratorText
: The text to be displayed in the item to remind the user of the accelerator.

XmNvalueChangedCallback
: The member function to be invoked when this menu item is selected.

XmNlabelString
: The label to be displayed in this menu item.

XmNmnemonic	The mnemonic that can be used to select this item.
XmNundoCallback	The optional member function that should be called if this item is reversed using the ViewKit undo mechanism.
XmNset	Determines whether this item is selected by default.

Menu Separator

The **MenuSeparator** corresponds to an **XmSeparatorGadget**, and is intended to be added to a menu pane as a decorative item to separate other entries.

MenuSeparator will be represented in the program as a ViewKit **VkMenuSeparator** object.

ConfirmFirst

The **ConfirmFirst** corresponds to an **XmPushButtonGadget**, and is intended to be added to a menu pane as a selectable command entry that asks the user for confirmation before executing the command. The MenuEntry will be represented in the program as a ViewKit **VkMenuConfirmFirstAction** object.

MenuToggle Resources

Following are the **MenuToggle** resources available through RapidApp:

XmNaccelerator	A description of the accelerator keys that can be used to invoke this menu item.
XmNacceleratorText	The text to be displayed in the item to remind the user of the accelerator.
XmNactivateCallback	The member function to be invoked when this menu item is selected.

XmNlabelString
The label to be displayed in this menu item.

XmNmnemonic
The mnemonic that can be used to select this item.

ViewKit Palette

The ViewKit palette (see Figure B-14) contains ViewKit interface elements such as tab panel, tick marks, and graph.

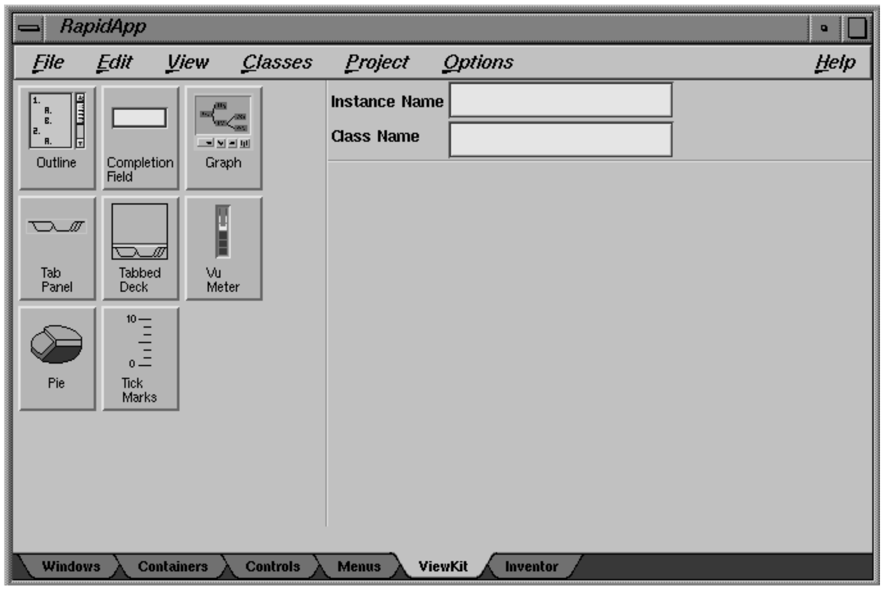


Figure B-14 ViewKit Palette

The user interface elements available through this palette are described in the following sections.

VkOutline

The **VkOutline** class allows you to display a tree of strings in an outline fashion. Each string is displayed in a line with an indentation proportional to its depth in the tree. Each non-leaf string has a control icon displayed to its left. The control icon denotes whether the subtree under the string is displayed (open) or not (closed). Control icons can be left-clicked by users to toggle between open and closed states.

This component cannot be manipulated via RapidApp, but can be added and positioned using RapidApp and manipulated programmatically. See the **VkOutline** reference page for details.

VkCompletionField

The **VkCompletionField** component is a text input field that supports name expansion. If the user types a space, the component attempts to complete the current contents of the text field, based on a known list of possible expansions. Applications must provide the list of possible expansions.

These can be provided programmatically, or they can be entered using RapidApp by providing a comma-separated list of strings as the **completionList** resource.

Applications that wish to be notified when you press `<Enter>` in the text field can register a ViewKit C++-style callback using the `VkCompletionFiled::enterCallback()` hook. This can only be done programmatically.

VkGraph

The **VkGraph** class is a component that provides a high-level interface to an underlying **SgGraph** widget. Graphs are constructed by specifying parent/child parents of objects, represented by the **VkNode** class. The **VkGraph** class constructs an abstract graph from these objects and allows applications or users to specify which portions of the graph to display at any one time. In this way, the **VkGraph** component supports graphs that can be larger than it is practical to display at one time.

Nodes must be created programmatically. A number of resources that affect either **VkGraph** or the underlying **SgGraph** widget can be set using RapidApp.

VkTabPanel

VkTabPanel presents a row or column of overlaid tabs. One tab is always selected and appears on top of all the others. The user can left-click on a tab to select it. When the tabs do not fit within the provided space, end-indicators appear as necessary to represent a set of collapsed tabs. When the user left-clicks or right-clicks in an end-indicator, a popup menu appears listing all the tabs. The user may choose an item to select the corresponding tab.

Tabs can be added programmatically, or they can be entered as a comma-separated set of strings in the “tabs” resource input area.

RapidApp currently supports only horizontal a orientation.

VkTabbedDeck

VkTabbedDeck is a composite component that combines a ViewKit **VkDeck** manager and a **VkTabPanel**. You can add items to the **VkTabbedDeck** by simply dropping them on the container. Each new child becomes a new panel in the deck, and automatically adds a new tab that allows the user to switch to that panel.

VkVUMeter

VkVUMeter presents a vertical set of segments as a meter display, similar to that used by hi-fi audio displays. Its value ranges from 0 to 110, with 0 showing the most segments and 110 showing the least.

VkPie

This class is derived from **VkMeter** and displays data in the same way as that class. Values are added programmatically, one at a time, and displayed by calling an **update()** member function. The range of values displayed can be specified by calling the **reset()** member function with a new value.

VkTickMarks

VkTickMarks presents a vertical set of tick marks. It is most commonly used next to a vertical **XmScale** widget. The tick marks can be right-justified with the labels to the left (the default), or left justified with the labels to the right. The former is used when the component is to the left of the scale, and the latter when the component is to the right.

Inventor Palette

The Inventor palette (see Figure B-15) contains Inventor interface elements such as material list, light slider set, and render area.

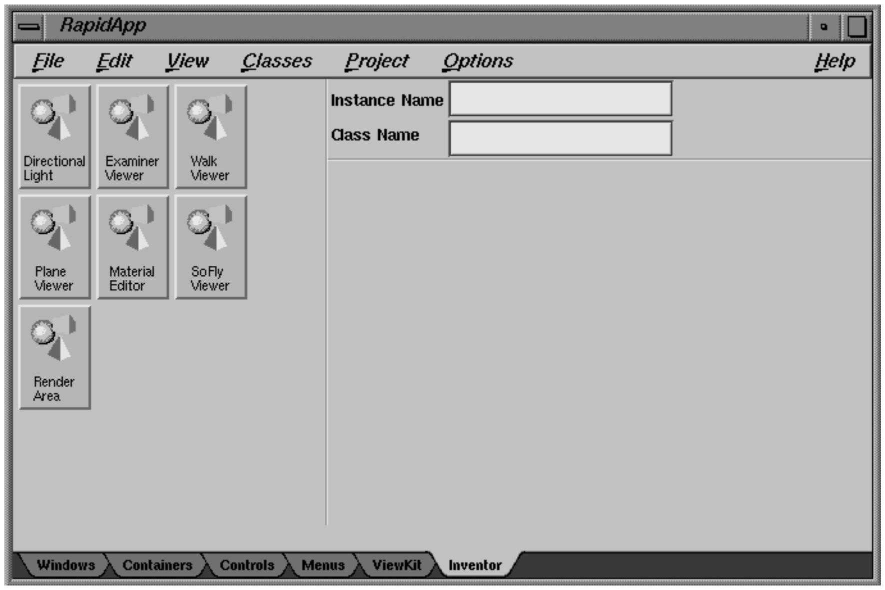


Figure B-15 Inventor Palette

The user interface elements available through this palette are described in the following sections.

Examiner Viewer

An Inventor scene viewer. The Examiner viewer component allows you to rotate the view around a point of interest using a virtual trackball. The viewer uses the camera focalDistance field to figure out the point of rotation, which is usually set to be at the center of the scene. In addition to allowing you to rotate the camera around the point of interest, this viewer also allows you to translate the camera in the viewer plane, as well as dolly (move forward and backward) to get closer to or further away from the point of interest. The viewer also supports seek to quickly move the camera to a desired object or point. See the reference page or the *Inventor Mentor* for more details.

Examiner Viewer Resources

Following are the **SoXtExaminerViewer** resources available through RapidApp:

animationEnabled	Enable or disable the spinning animation feature of the viewe.
antialiasing	Set the antialiasing for rendering. If this resource is set to True, “smoothing” is enabled. Smoothing uses OpenGL’s line- and point-smoothing features to provide cheap antialiasing of lines and points.
border	Toggles the border around the viewe ron or off
bufferingType	Sets the current buffering type.
decoration	Toggles the controls surrounding the viewer on or off.
drawStyle	Sets the current drawing style in the main view. See the SoXtViewer reference page for more details.
headlight	Turns the headlight on / off.
popupMenuEnabled	Activates or deactivates the right mouse button popup menu over the viewer.
sceneGraph	Specifies a filename of a scene graph to be displayed.

feedbackVisibility	Show / Hide the point of rotation feedback, which only appears while in viewing mode (default in off).
feedbackSize	Set the point of rotation feedback size in pixels (default 20 pix).

Walk Viewer

An Inventor scene viewer. The paradigm for this viewer is a walkthrough of an architectural model. Its primary behavior is forward, backward, and left / right turning motion while maintaining a constant “eye level”. It is also possible to stop and look around at the scene. The eye level plane can be disabled, allowing the viewer to proceed in the “look at” direction, as if on an escalator. The eye level plane can also be translated up and down ‹em similar to an elevator. See the reference page or the *Inventor Mentor* for more details.

Walk Viewer Resources

Following are the **SoXtWalkViewer** resources available through RapidApp:

antialiasing	Set the antialiasing for rendering. If this resource is set to True, “smoothing” is enabled. Smoothing uses OpenGL’s line- and point-smoothing features to provide cheap antialiasing of lines and points.
border	Toggles the border around the viewe ron or off
bufferingType	Sets the current buffering type.
decoration	Toggles the controls surrounding the viewer on or off.
drawStyle	Sets the current drawing style in the main view. See the SoXtViewer reference page for more details.
headlight	Turns the headlight on / off.
popupMenuEnabled	Activates or deactivates the right mouse button popup menu over the viewer.
sceneGraph	Specifies a filename of a scene graph to be displayed.

Plane Viewer

An Inventor scene viewer. The Plane viewer component allows the user to translate the camera in the viewing plane, as well as dolly (move foward / backward) and zoom in and out. The viewer also allows the user to roll the camera (rotate around the forward direction) and seek to objects which will specify a new viewing plane. This viewer could be used for modeling, in drafting, and architectural work. The camera can be aligned to the X, Y or Z axes. See the reference page or the *Inventor Mentor* for more details.

Plane Viewer Resources

Following are the **SoXtPlaneViewer** resources available through RapidApp:

border	Toggles the border around the viewe ron or off
bufferingType	Sets the current buffering type.
decoration	Toggles the controls surrounding the viewer on or off.
drawStyle	Sets the current drawing style in the main view. See the SoXtViewer reference page for more details.
headlight	Turns the headlight on / off.
popupMenuEnabled	Activates or deactivates the right mouse button popup menu over the viewer.
sceneGraph	Specifies a filename of a scene graph to be displayed.

Material Editor

This Inventor class is used to edit the material properties of an SoMaterial. node. The editor can also directly be used using callbacks instead of attaching it to a node. The component consists of a render area displaying a test sphere, some sliders, a set of radio buttons, and a menu. The sphere displays the current material being edited. There is one slider for each material coefficient. Those fields are ambient, diffuse, specular, emissive (all of which are colors); and transparency and shininess (which are scalar values). A color editor can be opened to edit the color slider base color. A material list displays palettes of predefined materials from which to choose.

The editor can currently be attached to only one material at a time. Attaching two different materials will automatically detach the first one before attaching the second. See the reference page or the *Inventor Mentor* for more details.

Directional Light

This Inventor class is used to edit an **SoDirectionalLight** node (color, intensity, and direction are changed). In addition to directly editing directional light nodes, the editor can also be used with callbacks which will be called whenever the light is changed. The component consists of a render area and a value slider in the main window, with controls to display a color picker. In the render area there appears a sphere representing the world, and a directional light manipulator representing the direction of the light. Picking on the manipulator and moving the mouse provides direct manipulation of the light direction. The color picker is used to edit the color, and the value slider edits the intensity. See the reference page or the *Inventor Mentor* for more details.

SoFly Viewer

This Inventor scene viewer is intended to simulate flight through space, with a constant world up direction. The viewer only constrains the camera to keep the user from flying upside down. No mouse buttons need to be pressed in order to fly. The mouse position is used only for steering, while mouse clicks are used to increase or decrease the viewer speed.

The viewer allows you to tilt your head up / down / right / left and move in the direction you are looking (forward or backward). The viewer also supports seek to quickly move the camera to a desired object or point. See the man page or the Inventor Mentor for more details.

SoFly Viewer Resources

Following are the **SoXtFlyViewer** resources available through RapidApp:

antialiasing	Set the antialiasing for rendering. If this resource is set to True, “smoothing” is enabled. Smoothing uses OpenGL’s line- and point-smoothing features to provide cheap antialiasing of lines and points.
border	Toggles the border around the viewe ron or off
bufferingType	Sets the current buffering type.
decoration	Toggles the controls surrounding the viewer on or off.
drawStyle	Sets the current drawing style in the main view. See the SoXtViewer reference page for more details.
headlight	Turns the headlight on / off.
popupMenuEnabled	Activates or deactivates the right mouse button popup menu over the viewer.
sceneGraph	Specifies a filename of a scene graph to be displayed.
viewing	Set whether the viewer is turned on or off. When turned on, events are consumed by the viewer. When viewing is off, events are processed by the viewer’s render area. This means events will be sent down to the scene graph for processing (in other words, picking can occur).

Render Area

This Inventor class provides Inventor rendering and event handling inside a GLX Motif widget. There is a routine to specify the scene to render. The scene is automatically rendered whenever anything under it changes (a data sensor is attached to the root of the scene), unless explicitly told not to do so (manual redraws). Users can also set Inventor rendering attributes such as the transparency type, antialiasing on or off, etc. This class employs a **SoSceneManager** to manage rendering and event handling. See the reference page or the *Inventor Mentor* for more details.

Render Area Resources

Following are the **SoXtRenderArea** resources available through RapidApp:

antialiasing	Set the antialiasing for rendering. If this resource is set to True, “smoothing” is enabled. Smoothing uses OpenGL’s line- and point-smoothing features to provide cheap antialiasing of lines and points.
border	Toggles the border around the viewe ron or off
bufferingType	Sets the current buffering type.
decoration	Toggles the controls surrounding the viewer on or off.
drawStyle	Sets the current drawing style in the main view. See the SoXtViewer reference page for more details.
headlight	Turns the headlight on / off.
popupMenuEnabled	Activates or deactivates the right mouse button popup menu over the viewer.
sceneGraph	Specifies a filename of a scene graph to be displayed.
viewing	Set whether the viewer is turned on or off. When turned on, events are consumed by the viewer. When viewing is off, events are processed by the viewer’s render area. This means events will be sent down to the scene graph for processing (in other words, picking can occur).

Source Code for the Calculator Application

This appendix lists and discusses some of the source files for the simple calculator application built in “Example: A Calculator” on page 24. The version of the calculator program in this appendix includes the **Calculator** component created in “Creating Components” on page 89.

The Calculator main.C File

The body of any program generated by RapidApp is very simple. Example C-1 lists the *main.C* file for the calculator application.

Example C-1 Calculator *main.C* File

```
//////////////////////////////////////////////////////////////////////
// This is a driver ViewKit program generated by RapidApp

//
// This program instantiates a ViewKit VkApp object and creates
// any main window objects that are meant to be shown at startup.
// There should rarely be a reason to modify this file.
// Make application-specific changes in the classes created
// by the main window classes
// Some applications may wish to change this code to instantiate
// a different application class, however.
//////////////////////////////////////////////////////////////////////
#include <Vk/VkApp.h>


// Headers for classes used in this program

#include "CalcWindowMainWindow.h"

void main ( int argc, char **argv )
{
    extern void InitEZ(void);
    InitEZ();

    VkApp       *app;

    app = new VkApp("Calculator", &argc, argv);

    VkSimpleWindow *calcWindow  = new CalcWindowMainWindow("calcWindow");
    calcWindow->show();

    app->run ();
}
```

This file simply instantiates an IRIS ViewKit **VkApp** class and then creates a **CalcWindowMainWindow** object before entering an event loop (the ***run()*** statement).

The CalcWindowMainWindow Class

The **CalcWindowMainWindow** class is a simple top-level IRIS ViewKit window class derived from **VkSimpleWindow**. This class provides the basic functionality of a shell widget and handles window manager interaction. You normally shouldn't edit this class's files, but it's worthwhile to see what the code does. Example C-2 lists the **CalcWindowMainWindow** header file, and Example C-3 lists the **CalcWindowMainWindow** source file.

Example C-2 The Calculator *CalcWindowMainWindow.h* File

```
//////////////////////////////////////////////////////////////
//
// Header file for CalcWindowMainWindow
//
//    This class is a ViewKit VkSimpleWindow subclass
//
// Normally, very little in this file should need to be changed.
// Create/add/modify menus using the builder.
//
// Try to restrict any changes to adding members below the
// "//---- End generated code section" markers
// Doing so will allow you to make chnages using the builder
// without losing any changes you may have made manually
//
//////////////////////////////////////////////////////////////
#ifndef CALCWINDOWMAINWINDOW_H
#define CALCWINDOWMAINWINDOW_H
```

```
#include <Vk/VkSimpleWindow.h>

//---- End generated headers


//---- CalcWindowMainWindow class declaration

class CalcWindowMainWindow: public VkSimpleWindow {

  public:

    CalcWindowMainWindow(const char * name );
    ~CalcWindowMainWindow();
    const char *className();
    virtual Boolean okToQuit();

    //---- End generated code section

  protected:



    // Classes created by this class

    class Calculator *_calculator;

    //---- End generated code section

  private:


    //---- End generated code section

};
#endif
```

Example C-3 The Calculator *CalcWindowMainWindow.C* File

```
//////////////////////////////////////////////////////////////
//
// Source file for CalcWindowMainWindow
//
//    This class is a ViewKit VkSimpleWindow subclass
//
//
```

```
// Normally, very little in this file should need to be changed.
// Create/add/modify menus using the builder.
//
// Try to restrict any changes to the bodies of functions
// corresponding to menu items, the constructor and destructor.
//
// Add any new functions below the "//--- End Generated Code"
// markers
//
// Doing so will allow you to make changes using the builder
// without losing any changes you may have made manually
//
// Avoid gratuitous reformatting and other changes that might
// make it difficult to integrate changes made using the builder
//////////////////////////////////////////////////////////////
#include "CalcWindowMainWindow.h"
#include <Vk/VkApp.h>
#include <Vk/VkFileSelectionDialog.h>
#include <Vk/VkSubMenu.h>
#include <Vk/VkRadioSubMenu.h>
#include <Vk/VkMenuItem.h>
#include "Calculator.h"
//---- End Generated Headers


//---- Class declaration

CalcWindowMainWindow::CalcWindowMainWindow(const char *name) : VkSimpleWindow (na
me)
{
    // Create the view component contained by this window

    _calculator= new Calculator("calculator",mainWindowWidget());


    XtVaSetValues ( _calculator->baseWidget(),
                    XmNwidth, 289,
                    XmNheight, 201,
                    (XtPointer) NULL );

    // Add the component as the main view

    addView (_calculator);
    _calculator->setParent(this);
    //---- End Generated Code Section

} // End Constructor


CalcWindowMainWindow::~CalcWindowMainWindow()
{
    delete _calculator;
} // End destructor


const char *CalcWindowMainWindow::className()
{
    return ("CalcWindowMainWindow");
} // End className()


Boolean CalcWindowMainWindow::okToQuit()
{

    // This member function is called when the user quits by calling
    // theApplication->terminate() or uses the window manager close protocol
    // This function can abort the operation by returning FALSE, or do some.
    // cleanup before returning TRUE. The actual decision is normally passed on
    // to the view object


    // Query the view object, and give it a chance to cleanup

    return ( _calculator->okToQuit() );
} // End okToQuit()


//--- End generated member functions
```

Note the “End Generated Code Section” markers. If you do need to modify this class, you should do so below these markers only.

CalcWindowMainWindow declares the pointer to the **Calculator** component that it creates as a protected data member. this allows you to access the **Calculator** component in any member functions that you add to this class.

The **CalcWindowMainWindow** constructor calls the **VkSimpleWindow** constructor and then instantiates a **Calculator** object. After setting the initial size of the component, the constructor adds the **Calculator** object as a view of the window.

The **CalcWindowMainWindow** destructor deletes the **Calculator** object created by the window.

The ***className()*** member function is a "boilerplate" function that all IRIS ViewKit components must implement to support X resource management.

Before exiting, the **VkApp** class calls the ***okToQuit()*** member function for each top-level window in the program. This gives a program a chance to clean up (for example, closing databases) or abort the shutdown if necessary. The ***CalcWindowMainWindow::okToQuit()*** member function that RapidApp generates simply calls the ***okToQuit()*** function of the **Calculator** component.

The Calculator Class

Calculator is the user-defined class you created in RapidApp. RapidApp automatically places most of the user interface code the base class, **CalculatorUI**, so the **Calculator** class itself is very simple. The class header, shown in Example C-4, declares constructors, destructors, and a virtual function, ***add()***, which is the function called when the user presses the "=" button on the calculator interface.

Example C-4 The Calculator *Calculator.h* File

```
/////////////////////////////////////////////////////////////
//
// Header file for Calculator
//
//    This file is generated by RapidApp
//
//    This class is derived from CalculatorUI which
//    implements the user interface created in
//    the interface builder. This class contains virtual
//    functions that are called from the user interface.
//
//    When you modify this header file, limit your changes to adding
//    members below the "//--- End generate code" markers
```

```
//
//    This will allow the builder to integrate changes more easily
//
//    This class is a ViewKit user interface "component".
//    For more information on how components are used, see the
//    "ViewKit Programmers' Manual", and the RapidApp
//    User's Guide.
//////////////////////////////////////////////////////////////
#ifndef CALCULATOR_H
#define CALCULATOR_H
#include "CalculatorUI.h"

#include <Vk/VkSimpleWindow.h>
//---- End generated headers


//---- Calculator class declaration

class Calculator : public CalculatorUI
{

  public:

    Calculator(const char *, Widget);
    Calculator(const char *);
    ~Calculator();
    const char *  className();
    virtual void setParent(VkSimpleWindow  *);
    //---- End generated code section


  protected:


    // These functions will be called as a result of callbacks
    // registered in CalculatorUI

    virtual void add ( Widget, XtPointer );

    VkSimpleWindow * _parent;
    //---- End generated code section


  private:
};
#endif
```

The *Calculator.C* source file consists primarily of empty functions. Most of the work is done by the **CalculatorUI** class. The listing shown in Example C-5 displays in bold the code you added to implement the class's functionality. This consists of changes to the ***add()*** function and two additional header files.

Example C-5 The Calculator *Calculator.C* File

```
/////////////////////////////////////////////////////////////
//
// Source file for Calculator
//
//    This file is generated by RapidApp
//
//    This class is derived from CalculatorUI which
//    implements the user interface created in
//    the interface builder. This class contains virtual
//    functions that are called from the user interface.
//
//    When you modify this source, limit your changes to
//    modifying the emtpy virtual functions. You can also add
//    new functions below the "//--- End generate code" markers
//
//    This will allow the builder to integrate changes more easily
//
//    This class is a ViewKit user interface "component".
//    For more information on how components are used, see the
//    "ViewKit Programmers' Manual", and the RapidApp
//    User's Guide.
/////////////////////////////////////////////////////////////

#include "Calculator.h"
#include <Vk/VkEZ.h>
#include <Xm/BulletinB.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/Separator.h>
#include <Xm/TextF.h>
#include <Vk/VkResource.h>
#include <Vk/VkSimpleWindow.h>

extern void VkUnimplemented(Widget, const char *);
```

```
//---- End generated headers

#include <Vk/VkFormat.h>
#include <stdlib.h>

///////////////////////////////////////////////////////////////////////////////
// The following non-container widgets are created by CalculatorUI and are
// available as protected data members inherited by this class
//
//  XmPushButton                    _button
//  XmSeparator             _separator
//  XmLabel                 _label
//  XmTextField             _result
//  XmTextField             _value2
//  XmTextField             _value1
//
///////////////////////////////////////////////////////////////////////////////


//---- Calculator Constructor

Calculator::Calculator(const char *name, Widget parent) :
                   CalculatorUI(name, parent)
{
    // This constructor calls CalculatorUI(parent, name)
    // which calls CalculatorUI::create() to create
    // the widgets for this component. Any code added here
    // is called after the component's interface has been built

    //--- Add application code here:


} // End Constructor



Calculator::Calculator(const char *name) :
                   CalculatorUI(name)
 {
    // This constructor calls CalculatorUI(name)
    // which does not create any widgets. Usually, this
    // constructor is not used

    //--- Add application code here:
} // End Constructor



Calculator::~Calculator()
{
    // The base class destructors are responsible for
    // destroying all widgets and objects used in this component.
    // Only additional items created directly in this class
    // need to be freed here.

    //--- Add application destructor code here:


}


const char * Calculator::className() // classname
{
    return ("Calculator");
} // End className()


void Calculator::add ( Widget w, XtPointer callData )
{
    XmAnyCallbackStruct *cbs = (XmAnyCallbackStruct*) callData;

    //--- Comment out the following line when Calculator::add is implemented:

    //::VkUnimplemented ( w, "Calculator::add" );


    //--- Add application code for Calculator::add here:

    int a = atoi(XmTextFieldGetString(_value1));
    int b = atoi(XmTextFieldGetString(_value2));
    XmTextFieldSetString(_result, (char *) VkFormat("%d", a + b));

} // End Calculator::add()


void Calculator::setParent( VkSimpleWindow  * parent )
{
    // Store a pointer to the parent VkWindow. This can
    // be useful for accessing the menubar from this class.


    _parent = parent;

} // End Calculator::setParent()
```

The CalculatorUI Class

The **CalculatorUI** class contains all the code required to create the user interface. These files are rather long, so aren't listed in this appendix. Normally, you shouldn't change the header or source files for this class. Almost everything you might want to do can be handled in the derived class or by using RapidApp.

The Calculator Resource File

The *Calculator* file contains the default values for various X resources used by the calculator application. Example C-6 lists the calculator resource file. You typically shouldn't need to edit this file.

Example C-6 The Calculator Resource File

```
!
! Generated by Silicon Graphic's RapidApp.
!
!
! RapidApp 1.0
!
!
!
!Activate schemes and sgi mode by default
!
Calculator*useSchemes: all
Calculator*sgiMode: true
!
!SGI Style guide specifies pointer focus for applications
!
Calculator*keyboardFocusPolicy: pointer

Calculator*calcWindow.title: Calculator
Calculator*label.labelString: +
Calculator*button.labelString: =

!
! The following resources are for the classes
! and instances of classes.
!

!!-- End Generated Defaults
```

Makefile

The *Makefile* follows Silicon Graphics conventions. The Makefile also uses a few simple conventions that make the *Makefile* easier to maintain from within RapidApp.

Example C-7 The Calculator *Makefile* file

```
#!smake
#
# Makefile for calculator
# Generated by RapidApp
#
# This makefile follows various conventions used by SGI makefiles
# See RapidApp User’s Guide for more information about
# commondefs, commonrules and other SGI conventions
#
include $(ROOT)/usr/include/make/commondefs
#
# Local Definitions
#

# Directory in which inst images are placed

IMAGEDIR= images


#  The GL library being used, if needed

GLLIBS=
COMPONENTLIBS=


#
# The ViewKit stub help library (-lvkhelp) provides a simple
# implementation of the SGI help API. Changing this to -ldesktopUtil
# switches to the full IRIS Insight help system
#

HELPLIB= -lvkhelp

# Standard ViewKit header and libraries

VIEWKITFLAGS= -I$(ROOT)/usr/include/Vk
TOOLTALKLIBS=
NETLS=

EZLIB = -lvkEZ
VIEWKITLIBS= $(TOOLTALKLIBS) $(EZLIB) -lvk $(HELPLIB) $(NETLS) -lSgm -lXpm

# Local C++ options.
# woff 3262 shuts off warnings about arguments that are declared
# but not referenced.

WOFF= -woff 3262

LCXXOPTS = -nostdinc -I$(ROOT)/usr/include $(SAFLAG) $(WOFF) $(VIEWKITFLAGS)

# Add Additional libraries to USERLIBS:

USERLIBS=

LLDLIBS =  -L$(ROOT)/usr/lib $(USERLIBS) $(COMPONENTLIBS) $(VIEWKITLIBS) $(GLLIBS
) -lXm -lXt -lX11 -lgen

# While developing, leave OPTIMIZER set to -g.
# For delivery, change to -O2

OPTIMIZER= -g

# SGI makefiles don't recognize all C++ sufixes, so set up
# the one being used here.

CXXO3=$(CXXO2:.C=.o)
CXXOALL=$(CXXO3)

#
# Source Files generated by the builder. If files are added
# manually, add them to USERFILES
#
```

```
BUILDERFILES =  main.C\
                CalcWindowMainWindow.C\
                Calculator.C\
                CalculatorUI.C\
                unimplemented.C\
                $(NULL)

#
# Add any files added outside the builder here
#

USERFILES =

C++FILES = $(BUILDERFILES) $(USERFILES)


#
# The program being built
#

TARGETS=calculator
APPDEFAULTS=Calculator
default all: $(TARGETS)


$(TARGETS): $(OBJECTS)
        $(C++) $(OPTIMIZER) $(OBJECTS) $(LDFLAGS) -o $@

unimplemented.o: unimplemented.C
        $(C++) -c -O unimplemented.C
#
# These flags instruct the compiler to output
# analysis information for cvstatic
# Uncoment to enable
# Be sure to also disable smake if cvstatic is used

#SADIR= Calc.cvdb
#SAFLAG= -sa,$(SADIR)
#$(OBJECTS):$(SADIR)/cvdb.dbd
#$(SADIR)/cvdb.dbd :
#        [ -d $(SADIR) ] || mkdir $(SADIR)
#        cd $(SADIR); initcvdb.sh

#LDIRT=$(SADIR) vista.taf
```

```
#
# To install on the local machine, do 'make install'
#

install: all
        $(INSTALL) -F /usr/lib/X11/app-defaults Calculator
        $(INSTALL) -F /usr/sbin calculator

#
# To create inst images, do 'make image'
# An image subdirectory should already exist
#

$(IMAGEDIR):
        @mkdir $(IMAGEDIR)
image: $(TARGETS) $(IMAGEDIR)
        /usr/sbin/gendist -rbase / -sbase / -idb calculator.idb \
         -spec calculator.spec \
        -dist /usr/people/kenj/sgdx/Calc/images  -all

include $(COMMONRULES)
```

You shouldn't modify most of the *Makefile,* however the following areas are safe for you to change:

- The variable IMAGEDIR controls the location at which installable images are generated. The default is a subdirectory of the current directory called *images.*
- If you wish to use the Developer Magic Static Analyzer on your application, need to uncomment the lines that generate the static analysis database for your application. You can do this manually or, from within RapidApp, you can:
    1. Select "Application" from the Options menu.
    2. In the Application Names dialog that appears, toggle on the Create Static Analysis Database option.
    3. Select "Generate C++" from the Project menu to regenerate the *Makefile.*
- To add source files to the *Makefile* that you create outside of RapidApp, simply list them after the USERFILES variable; they will be compiled the next time you build your application.

Appendix D

RapidApp Makefile Conventions

RapidApp uses several macros found in */usr/include/make* to generate a simple, easy-to-use *Makefile*. In many cases, you can use the *Makefile* generated by RapidApp without change. Occasionally, you might need to add files and libraries to the *Makefile*.

To add files, simply add them to the USERFILES variable, which RapidApp generates as an empty list. RapidApp lists the code files that it generates in the BUILDERFILES variable; you shouldn't edit this list. The *Makefile* concatenates USERFILES and BUILDERFILES and assigns the result to C++FILES, which the *Makefile* uses to build the program according to the built-in rules.

RapidApp automatically lists in the *Makefile* the libraries it requires to compile the interface code for your program. However, you might need to add additional libraries to the link line to support the functionality you added to your program. To do this, list the libraries in the USERLIBS variable, which RapidApp generates as an empty list.

Conventions Used In This Makefile

Nearly all paths referenced directly or indirectly in the Makefile are qualified by the variable ROOT. By default, if this variable isn't set, the paths are relative to / (the root directory). However, setting this variable allows you to point to an alternate set of development libraries, compilers, and other tools. Typically, you don't need to change this variable.

The *Makefile* loads many definitions with the line near the top of the file:

```
include $(ROOT)/usr/include/make/commondefs
```

You can browse this file if you are interested in the symbols defined, but the following are the most useful definitions that you should know about:

DIRT	Includes files like *core*, **.o*, and so on. You can add to this list by setting listing the files in the LDIRT variable in your *Makefile*. All items listed in DIRT are removed when you execute **`make clean`**.
C++FLAGS	Determines the flags passed to the C++ compiler. You can add to these flags by defining an LC++FLAGS variable in your *Makefile*.
LOCALDEFS and LOCALRULES	The definition of these variables cause the *Makefile* to check for files in your directory named *.localdefs* or *.localrules* and, if they exist, load them after it loads all the standard definitions and rules. This provides an easy way to extend the *Makefile* without modifying it heavily.

Most options you would normally set in a *Makefile* are available as symbols defined directly in the *Makefile*, and should be understandable by reading the comments in the *Makefile*. For example, to prepare your program for production by compiling with the optimizer on, change the line

```
OPTIMIZER= -g
```

to

```
OPTIMZER=-O2
```

The last line of the Makefile includes a common set of rules. The path represented by the COMMONRULES variable is defined in the *commondefs* file. This path is typically */usr/include/make/commonrules*.

Among the rules defined in */usr/include/make/commonrules* are:

make clean	Removes “dirt”, as defined by the DIRT variable
make clobber	Removes targets, dirt, and Makedepend files
make rmtargets	Removes targets only

VkEZ Reference

This section provides details about the VkEZ utility. There are two main features of VkEZ:

- General operators that are applied directly to the object. For example:

  ```
  EZ(widget) = "a label";
  ```

- Operators that are applied to an attribute (resource) supported by the widget. For example:

  ```
  EZ(widget).foreground = "red";
  ```

This appendix describes the VkEZ operators. The description of each operator lists the widgets which support that operator and defines how the operator works on each widget.

General Operators

operator String()

This operator returns a character string from the widget. For example:

```
strcpy(buffer, EZ(text));
```

This example copies the contents of a text widget into *buffer*.

The following list describes the behavior of this operator for each widget that supports it.

XmLabel, XmLabelGadget and subclasses
: returns the current value of the XmNlabelString resource as a character string

XmText, XmTextField	returns the contents of the text widget
XmList	returns the text associated with the currently selected item

operator int()

This operator returns an integer value from the widget. For example:

```
int value = EZ(dial);
```

This example assigns the current value of a dial widget to *value*.

The following list describes the behavior of this operator for each widget that supports it.

XmToggleButton and XmToggleButtonGadget	returns 1 if set, 0 if not set
XmScrollbar, XmScale	returns the current position of the slider
SgDial	returns the current position of the pointer
XmText, XmTextField	returns the result of calling atoi(3C) on the current contents of the text widget
XmList	returns the currently selected position

Assignment Operators

```
EZ& operator=(int);
EZ& operator=(float);
EZ& operator=(const char *);
```

These operators assign integer, floating point, and character values to a widget. For example:

```
 EZ(text) = 12345;
```

This example displays the integer 12345 in a text field.

The following list describes the behavior of the integer assignment operator for each widget that supports it.

XmToggle, XmToggleButtonGadget	if the specified value is zero, turns the toggle off; if the specified value is non-zero value turns the toggle on
XmScrollbar, XmScale	sets the current slider position to the specified value
SgDial	sets the current pointer position to the specified value
XmLabel, XmLabelGadget and subclasses (except toggle)	displays the specified value as the XmNlabelString resource
XmText, XmTextField	displays the specified value as the XmNvalue resource
XmList	sets the current position index to the specified value

The following list describes the behavior of the floating point assignment operator for each widget that supports it.

XmScrollbar, XmScale	sets the current slider position to the integer equivalent of the specified value (the floating point value is truncated)
SgDial	sets the current pointer position to the integer equivalent of the specified value (the floating point value is truncated)
XmLabel, XmLabelGadget and subclasses	displays the specified floating point value as the XmNlabelString resource
XmText, XmTextField	displays the specified floating point value as the XmNvalue resource
XmList	sets the current position index to the integer equivalent of the specified value (the floating point value is truncated)

The following list describes the behavior of the character string assignment operator for each widget that supports it.

XmScale	sets the title to the specified string

XmLabel, XmLabelGadget and subclasses
: displays the specified string as the XmNlabelString resource

XmText, XmTextField
: displays the specified string as the XmNvalue resource

XmList
: treats the specified string as comma-separated list of items and sets the list widget to display the new items, removing any old contents

Append Operators

```
EZ& operator+=(int);
EZ& operator+=(float);
EZ& operator+=(const char *);
EZ& operator<<(int);
EZ& operator<<(float);
EZ& operator<<(const char *);
```

These operators append the right side expression to the current value of a widget. Logically, the += operators make more sense for numerical operations, while the << operators seem more suitable for strings, but they are actually equivalent and either can be used.

The following list describes the behavior of the integer and floating point append operator for each widget that supports them.

XmToggle, XmToggleButtonGadget
: increments the current value of XmNset by the specified value, so

```
EZ(toggle) += 0;
```

does nothing, while

```
EZ(toggle) +=1;
```

sets *toggle* if it is not already set

XmScale, XmScrollbar
: increases the position of the slider by the specified value

SgDial
: increases the position of the pointer by the specified value

XmLabel, XmLabelGadget and subclasses	appends the specified value to the current XmNlabelString resource
XmText, XmTextField	appends the specified value to the current XmNvalue resource
XmList	increments the current position index by the specified value

The following list describes the behavior of the character string append operator for each widget that supports it.

XmLabel, XmLabelGadget and subclasses	appends the given string to the current value of the XmNlabelString resource
XmScale	appends the give string to the current value of the XmNtitle resource
XmText, XmTextField	appends the value to the XmNvalue resource
XmList	treats the string as a comma-separated list of items and adds the items to the list widget's current contents

Decrement Operator

```
EZ& operator-=(int);
```

This operator decrements the current value associated with a widget. This operator has more limited use than the += operator.

The following list describes the behavior of the decrement operator for each widget that supports it.

XmToggle, XmToggleButtonGadget

decrements the current value of XmNset by the specified value, so

```
EZ(toggle) -= 0;
```

does nothing, while

```
EZ(toggle) -= 1;
```

	unsets *toggle* if it is not already unset
XmScale, XmScrollbar	decreases the position of the slider by the specified value
SgDial	decreases the position of the pointer by the specified value
XmList	decrements the current position index by the specified value

Attributes

This section lists attributes which can be modified using VkEZ. Each of these attributes can be retrieved or set. For example:

```
int width = EZ(widget).width;
EZ(widget).width = 20;
EZ(widget).foreground = "red";
Pixel index = EZ(widget).background;
```

The following attributes are supported by all widgets:

border, width, height, x, y

The following attributes are supported by all widgets that have setting support for Pixel or char *:

background, foreground

The following attributes are supported by XmLabel, XmLabelGadget and subclasses:

label

The following attributes supported by XmScale, XmScrollBar, SgDial:

value, minimum, maximum

Glossary

attachment icons

Symbols displayed on an interface element when contained by a form. The attachment icons allow you to edit the attachment constraints interactively. See also *interface elements, containers,* and *constraints.*

cascade buttons

Push buttons that, when the user clicks them, display pulldown menus.

child elements

The interface elements contained or grouped by a container widget. See also *interface elements* and *widgets.*

components

Interface elements based on IRIS ViewKit classes. A component is a C++ class and can contain several other components and / or widgets. See also *interface elements* and *widgets.* See also *interface elements* and *widgets.*

constraints

Resources added to an interface element by a container that affect the element's position within the container. See also *interface elements, containers,* and *resources.*

containers

Widget that can group or contain other interface elements. See also *interface elements* and *widgets.*

co-primary windows

Top-level windows within an application used for major data manipulation or viewing of data outside of the main window. See also *main windows.*

elements

See *interface elements*.

interface elements

Any objects that you create, select, position, and manipulate in RapidApp. Interface elements can be either *components* or *widgets*.

IRIS IM

The Silicon Graphics port of the industry-standard OSF / Motif interface toolkit.

main windows

The application’s main controlling window used to view or manipulate data, get access to other windows within the application, and quit the application. There should be only one main primary window per application. See also *co-primary windows*.

radio behavior

The behavior of a group of toggles where only one toggle at a time can be active. When the user toggles on a button in the group, any other toggle in the group that was on turns off.

reparenting

Moving an interface element from one container widget to another. See also *interface elements* and *widgets*.

resources

Attributes of interface elements that change their appearance or behavior. See also *interface elements*.

snap grid

An invisible grid to which interface elements “snap” when you move or resize them. See also *interface elements*.

widgets

Interface components that are part of the IRIS IM toolkit. See also *interface elements* and *IRIS IM*.

Index