

Developer Magic™: RapidApp™ User's Guide

Document Number 007-2590-004

CONTRIBUTORS

Written by Doug Young, Terri Wanke, and Ken Jones
Production by Linda Rae Sande
Engineering contributions by Doug Young

© 1995, 1996, Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Silicon Graphics, the Silicon Graphics logo, IRIS, and OpenGL are registered trademarks and CASEVision, Developer Magic, Indigo Magic, Inventor, IRIS IM, IRIS InSight, IRIS Showcase, IRIS ViewKit, Open Inventor, and RapidApp are trademarks of Silicon Graphics, Inc. Builder Xcessory is a trademark of Integrated Computer Systems, Inc. Ada is a registered trademark of Ada Joint Program Office, U.S. Government. Motif and OSF/Motif are registered trademarks of Open Software Foundation. X Window System is a trademark of Massachusetts Institute of Technology. ToolTalk is a trademark of Sun Microsystems, Inc. NetLS is a trademark of Apollo Computer, Inc., a subsidiary of Hewlett-Packard Company.

Developer Magic™: RapidApp™ User's Guide
Document Number 007-2590-004

Contents

List of Examples xix

List of Figures xxi

About This Guide xxv

What This Guide Contains xxv

What You Should Know Before Reading This Guide xxvi

Suggested Reading xxvii

Font Conventions in This Guide xxx

Upgrading to Builder Xcessory xxxi

1. **RapidApp Tour** 1
 - Building a Calculator: A RapidApp Example 1
2. **RapidApp Basics** 11
 - RapidApp Overview 11
 - Prerequisites for Installing RapidApp 13
 - Setting Up Your Working Environment for RapidApp 13
 - Project Directory 13
 - Resource Files 14
 - Editor 14
 - Color-Map 14
 - Starting RapidApp 15
 - The Startup Screen 15
 - RapidApp Tips 16

- The RapidApp Interface 17
 - Menu Bar 18
 - Palette 18
 - Instance Header 19
 - Resource Editor 19
 - Quick Help 19
 - Palette Tabs 19
- 3. Interface Design 21**
 - The Indigo Magic Style 21
 - Basic Interface Construction 22
 - See Also 23
 - Parent/Child Interaction 23
 - Parent/Child Relationship 23
 - Selecting a Parent Element 24
 - Confining Element Creation to the Selected Parent 24
 - Interface Element Creation 24
 - Creating Interface Elements 25
 - Copying Interface Elements 28
 - Aborting Interface Element Creation 28
 - See Also 28

Interface Element Manipulation	29
Moving and Resizing Interface Elements	29
Moving and Resizing Windows	29
Moving and Resizing Containers	29
Moving and Resizing Elements within Containers	30
Moving Elements to Another Container	35
Deleting Interface Elements	36
Naming Interface Elements	36
Modifying an Interface Element's Appearance or Behavior	37
Modifying Resources	37
Callback Resources	38
Constraint Resources	39
Dynamic Resources	39
Extended Resources	40
Additional Interaction Techniques	40
Locking on to an Element	40
Viewing the Interface Element Hierarchy	40
Save Your Interface and Files	42
4. Application Development	43
Development Cycle	43
Test Your Application	44
Browsing Source	44
Create an Application Icon	45
Create a Minimized Window Icon	46
Create Installable Images	46
Development Model	47
Object-Oriented Components	47

- 5. **Code Creation and Management** 49
 - Generate Code 49
 - Generating RapidApp's Code 49
 - Adding and Editing Code 50
 - Following a Few Simple Coding Rules 50
 - Using Colors and Fonts 51
 - Using the Debugger to Add Code 51
 - Using a Text Editor to Add Code 54
 - Using EZ Convenience Functions 54
 - Code Management 55
 - Code Generation 55
 - Class Architecture 57
 - Code Merging 58
 - Block Merge** 58
 - Three-Way Merge** 59
- 6. **Advanced Topics** 63
 - Work With Windows 63
 - Simple Windows 65
 - VkWindows 65
 - Main Primary and Co-Primary Windows 69
 - Adding New Top-Level Window Layouts to RapidApp 70
 - Work With Containers 72
 - Bulletin Board 73
 - Rubber Board 74
 - Spring Box 79
 - Form 82
 - Paned Windows 86
 - RowColumn 88
 - Radio Box 89
 - Frame 89
 - Scrolled Window 90
 - Drawing Areas 90
 - Tabbed Deck 91

Work With Menus	92
Menu Bars	92
Creating a Menu Bar	93
Adding Panes to a Menu Bar	93
Removing Panes From a Menu Bar	94
Moving Panes In a Menu Bar	94
Menu Panes	94
Displaying and Hiding a Menu's Contents	94
Adding Items to a Menu	95
Moving Items in a Menu	96
Removing Items From a Menu	96
Option Menus	96
Work With Dialogs	97
Using the IRIS ViewKit Dialog System for Standard Dialogs	97
Creating and Using Custom Dialogs	99
Selecting a Dialog to Customize	99
Adding Elements to the Dialog	100
Adding a Menu Bar to the Dialog	100
Setting the Dialog Message, Dialog Title, and Button Labels	100
Code Generation for Customized Standard Dialogs	100
Code Generation for Dialogs Customized From the Dialog Window	101
Example of a Custom Dialog	102
Work With User-Defined Components	104
Creating Components	105
An Example: Creating a Component	107
Deriving From Your Own Super Class	109
Using Components	110
Editing Components	111
Deleting Components	111
7. VkeZ Library	113
EZ Convenience Functions	113
Examples Using the EZ Functions	115
Support for Widget Resources	115

- VkEZ Operators 116
 - General Operators 116
 - String() 116
 - int() 117
 - Assignment Operators 117
 - Append Operators 119
 - Decrement Operator 120
 - Attributes 121
- 8. Component Libraries 123**
 - Work With Component Libraries 123
 - Introduction 123
 - Configuring RapidApp 124
 - Build a Component Library 125
 - Generating Code for the Component Library 125
 - Building the Component Library 126
 - Example of Creating a Component Library 127
 - Install a Component Library 127
 - Package a Component Library for Distribution 127
 - Load Components Onto Palettes 128
 - Introduction 128
 - Loading Components Into RapidApp 129
 - Loading a Component Onto a RapidApp Palette 129
 - Example of Loading a Component Onto a RapidApp Palette 133
 - Files Generated When Loading Components on a Palette 134
 - Deleting a Component from a Custom RapidApp Palette 135
 - Creating an Installable Image 136
 - Adding Resources to Components 136
 - Arguments and Argument Types 141
 - Support for Callbacks 142

	Load Non-RapidApp Components	143
	Introduction	143
	Requirements for Loading Components Not Created With RapidApp	144
	Testing Components	147
	Adding Custom Base Classes to RapidApp	149
9.	Example Applications	151
	A Simple Open Inventor Program	151
	Online Examples	158
A.	RapidApp Reference	161
	Reference	161
	Global Objects	162
	File Menu	162
	Edit Menu	163
	View Menu	164
	Classes Menu	165
	Project Menu	165
	RapidApp Preferences Dialog	166
	Project Card	167
	RapidApp Card	169
	Code Style Card	171
	Merge Options Card	173
	RapidApp Component Importer Dialog	175
	Palette Tabs	176
	Keys and Shortcuts	176

Windows Palette	178
Simple Window	179
VkSimpleWindow Resources	179
VkWindow	180
Dialog Window	180
File Dialog	181
Info Dialog	181
Error Dialog	182
Warning Dialog	182
Question Dialog	182

Containers Palette	183
Bulletin Board	184
Bulletin Board Resources	184
Rubber Board	185
Rubber Board Resources	185
Spring Box	185
Spring Box Resources	186
Spring Box Constraint Resources	188
Form	188
Form Resources	189
Form Constraint Resources	190
Paned and HPaned Windows	191
Paned and HPaned Window Resources	192
Paned and HPaned Window Constraint Resources	193
Row Column	193
Row Column Resources	194
Scrolled Window	195
Scrolled Window Resources	196
Drawing Area and Visual Drawing	196
Drawing Area and Visual Drawing Resources	197
Visual Drawing Resources	197
Radio Box	198
Radio Box Resources	198
Frame	198
Frame Resources	199
Frame Constraint Resources	199
GLDraw	200
Tabbed Deck	203
Tabbed Deck Resources	203

- Controls Palette 204
 - Push Button 204
 - Push Button Resources 205
 - Code Examples 206
 - Toggle Button 207
 - Toggle Button Resources 208
 - Code Examples 210
 - Drawn Button 210
 - Drawn Button Resources 210
 - Arrow Button 212
 - Resources 212
 - Label 213
 - Label Resources 213
 - Code Examples 214
 - Separator 215
 - Separator Resources 215
 - Scroll Bar 216
 - Scroll Bar Resources 216
 - Code Examples 217
 - Scale 217
 - Scale Resources 218
 - Code Examples 219
 - Scrolled List 219
 - Scrolled Window Resources 221
 - List Resources 221
 - Scrolled Text 223
 - Scrolled Text Resources 223
 - Text Field 224
 - Text Field Resources 224
 - Finder 225
 - Finder Resources 226
 - Thumb Wheel 226
 - Thumb Wheel Resources 226

Dial	227
Dial Resources	228
LED Button	229
LED Button Resources	229
Drop Pocket	231
Drop Pocket Resources	231
Menus Palette	232
Pulldown Menu	232
Pulldown Resources	233
Cascade Menu	233
Cascade Resources	233
Radio Pulldown	234
RadioPulldown Resources	234
OptionsMenu	234
OptionsMenu Resources	234
Menu Entry	235
MenuEntry Resources	235
Menu Label	235
MenuLabel Resources	235
Menu Toggle	236
MenuToggle Resources	236
Menu Separator	236
Confirm First	237
MenuToggle Resources	237
Menu Bar	237
ViewKit Palette	238
VkOutline	238
VkCompletionField	239
VkGraph	239
VkPie	239
VkTabPanel	240
VkVUMeter	240
VkTickMarks	240

- Inventor Palette 241
 - Directional Light 241
 - Examiner Viewer 242
 - Examiner Viewer Resources 242
 - Walk Viewer 243
 - Walk Viewer Resources 243
 - Render Area 244
 - Render Area Resources 244
 - Plane Viewer 245
 - Plane Viewer Resources 245
 - Material Editor 245
 - SoFly Viewer 246
 - SoFly Viewer Resources 246

B.	RapidApp Makefile Conventions	249
C.	Frequently Asked Questions and Tips	251
	Frequently Asked Questions	251
	Bitmap/Pixmap Icons	253
	Code	254
	Client Data	254
	Leading Underscores	254
	Non-C++ Languages	254
	Portability	255
	User-Edited Code	255
	Containers	256
	Size	256
	Dialogs	256
	And Buttons	256
	Creation	256
	Editors	257
	Help	257
	Help System	257
	Hiding Help	258
	Interface Elements	258
	Access	258
	Alignment	259
	Appearance	259
	Creation	260
	Names	260
	Resources	261
	Selection	262
	Size	262
	User-Defined Widgets	262
	Inventor	262
	Delayed Creation	262
	Menus	263
	Grayed-Out Menu Items	263

- Manipulation 263
- RapidApp Files 265
 - Adding Company Standard Header 265
 - Makefile 265
- TCL Support 265
- Tips 265
 - Accessing 265
 - Removing 265
- User-Defined Classes 266
 - Class Connection 266
 - Dynamic Behavior 267
 - Editing 267
 - Functionality 267
 - Import 268
 - Instantiation 268
- UIKit 268
 - Accessing 268
 - Delayed Creation 269
- VkEZ and EZ Functions 269
- Windows 270
 - And Elements 270
 - Resizable Layouts 270
- RapidApp Tips 271
 - Displaying Tips 271
 - List of Tips 271

D.	Source Code for the Calculator Application	283
	The Calculator main.C File	283
	The CalcWindowMainWindow Class	284
	The CalculatorUI Class	290
	The Calculator Class	290
	The Calculator Resource File	298
	Makefile	299
	Index	307

List of Examples

Example 1-1	Sample of Default Code Generated for Pushbutton Callback	8
Example 1-2	Example of Adding Functional Code to a Pushbutton Callback	9
Example A-1	Retrieving Text From a Subclass of Label Using the IRIS IM API	207
Example A-2	Retrieving Text From a Subclass of Label Using the VkeZ API	207
Example A-3	Setting Text on a Subclass of Label Using the VkeZ API	207
Example A-4	Setting the Indicator State on a Toggle Button Without Invoking Callbacks	210
Example A-5	Setting the Indicator State on a Toggle Button and Triggering Callbacks	210
Example A-6	Retrieving Text From a Subclass of Label Using the IRIS IM API	214
Example A-7	Retrieving Text From a Subclass of Label Using the VkeZ API	214
Example A-8	Setting Text on a Subclass of Label Using the VkeZ API	215
Example A-9	Getting the Value of a Scroll Bar Using the IRIS IM API	217
Example A-10	Getting the Value of a Scroll Bar Using the VkeZ API	217
Example A-11	Setting the Value of a Scroll Bar Using the IRIS IM API	217
Example A-12	Setting the Value of a Scroll Bar Using the VkeZ API	217
Example A-13	Getting the Value of a Scale Using the IRIS IM API	219
Example A-14	Getting the Value of a Scale Using the VkeZ API	219
Example A-15	Setting the Value of a Scale Using the IRIS IM API	219
Example A-16	Setting the Value of a Scale Using the VkeZ API	219
Example D-1	Calculator <i>main.C</i> File	283
Example D-2	The Calculator <i>CalcWindowMainWindow.h</i> File	285
Example D-3	The Calculator <i>CalcWindowMainWindow.C</i> File	287
Example D-4	The Calculator <i>Calculator.h</i> File	291
Example D-5	The Calculator <i>Calculator.C</i> File	293
Example D-6	The Calculator Resource File	298
Example D-7	The Calculator <i>Makefile</i> File	299

List of Figures

Figure 1-1	Completed Calculator Application	1
Figure 1-2	RapidApp's Main Window	2
Figure 1-3	Initial Calculator Layout	3
Figure 1-4	Second Text Field and Label	3
Figure 1-5	All Widgets In Place	4
Figure 1-6	RapidApp Information Window	5
Figure 1-7	Project Card	6
Figure 1-8	Building the Calculator Application	7
Figure 1-9	Working Calculator Application	10
Figure 2-1	RapidApp's Startup Screen	15
Figure 2-2	RapidApp's Main Window	17
Figure 3-1	Constructing an Interface	22
Figure 3-2	Parent/Child Relationship	23
Figure 3-3	Creating a Simple Window	27
Figure 3-4	Moving an Interface Element	32
Figure 3-5	Resizing an Interface Element	33
Figure 3-6	Moving an Interface Element in a RowColumn	35
Figure 3-7	Changing an Element's Name in the Header Area	36
Figure 3-8	Push Button Resources	38
Figure 3-9	Adding a Callback	38
Figure 3-10	Constraint Resources	39
Figure 3-11	Window with Three Labels	41
Figure 3-12	Window with Colored Interface Element Hierarchy	41
Figure 5-1	VkUnimplemented Function	52
Figure 5-2	Example of an Undefined Callback Function	53
Figure 6-1	The RapidApp Windows Palette	64
Figure 6-2	Default Configuration of VkWindow Component	66

Figure 6-3	Setting the Window Type	69
Figure 6-4	RapidApp Containers Palette	72
Figure 6-5	Rubber Board: Initial Layout	75
Figure 6-6	Rubber Board: Preparing for Larger Layout	76
Figure 6-7	Rubber Board: Final Layout	77
Figure 6-8	Effect of Resizing the Final Rubber Board Layout	78
Figure 6-9	Vertical and Horizontal Spring Boxes	79
Figure 6-10	Setting Spring Resources	79
Figure 6-11	Spring Box Behavior With Modified Values	80
Figure 6-12	Push Button in a Form	82
Figure 6-13	Setting the Top Offset to Zero	83
Figure 6-14	Using the Popup to Set an Offset	83
Figure 6-15	Displaying the Attachment Menu	84
Figure 6-16	Push Button With a Right Attachment	84
Figure 6-17	Drawing an Attachment	85
Figure 6-18	HPaned Window Container	87
Figure 6-19	Typical RowColumn Layout	88
Figure 6-20	Radio Box With Toggle Button Children	89
Figure 6-21	Frame Widget	89
Figure 6-22	Tabbed Deck	91
Figure 6-23	RapidApp Menus Palette	92
Figure 6-24	Default Configuration of Dialog Window	99
Figure 6-25	Example of a Custom Dialog	102
Figure 6-26	Make Class Dialog	105
Figure 6-27	Creating a Calculator Class	107
Figure 6-28	Class Header	108
Figure 6-29	Calculator Icon from User-Defined Palette	108
Figure 8-1	The RapidApp Component Importer Dialog	130
Figure 8-2	Example of Specifying a Component to Load on a RapidApp Palette	133
Figure 8-3	LabeledText Component	138
Figure 8-4	Using the LabeledText Component	140
Figure 8-5	LabeledText Resources	141

Figure 8-6	The Component Tester	147
Figure 8-7	The Component Tester Load Component Dialog	148
Figure 9-1	Adding an Examiner Viewer	152
Figure 9-2	The Completed Open Inventor Component	153
Figure 9-3	The Open Inventor Interface Displaying a Scene	155
Figure A-1	RapidApp Main Window	162
Figure A-2	File Menu	162
Figure A-3	Edit Menu	163
Figure A-4	View Menu	164
Figure A-5	Snap To Grid Toggle	164
Figure A-6	Classes Menu	165
Figure A-7	Project Menu	165
Figure A-8	The Project Card	167
Figure A-9	The RapidApp Card	169
Figure A-10	The Code Style Card	171
Figure A-11	The Merge Options Card	173
Figure A-12	Palette Tabs	176
Figure A-13	Windows Palette	178
Figure A-14	Containers Palette	183
Figure A-15	Controls Palette	204
Figure A-16	Menus Palette	232
Figure A-17	ViewKit Palette	238
Figure A-18	Inventor Palette	241
Figure C-1	Example of a Row Column Container	281

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, “RapidApp Tour,” provides a tutorial to demonstrate the basic use of RapidApp.
- Chapter 2, “RapidApp Basics,” provides an overview of RapidApp and describes how to set up your working environment.
- Chapter 3, “Interface Design,” describes the process of designing the interface to your application.
- Chapter 4, “Application Development,” describes the process of developing an application.
- Chapter 5, “Code Creation and Management,” describes how to create and generate code using RapidApp, as well as giving details about RapidApp’s approach to code management.
- Chapter 6, “Advanced Topics,” provides detailed information about working with RapidApp’s advanced features, such as choosing the right containers for your interface, using dialogs, and creating menus.
- Chapter 7, “VkeZ Library,” provides details about the VkeZ library and its EZ convenience functions.
- Chapter 8, “Component Libraries,” describes how to create libraries of reusable components for RapidApp.

- Chapter 9, “Example Applications,” provides some example applications created with RapidApp.
- Appendix A, “RapidApp Reference,” is a reference to RapidApp’s menus and palettes.
- Appendix B, “RapidApp Makefile Conventions,” documents the format of the *Makefile* the RapidApp generates.
- Appendix C, “Frequently Asked Questions and Tips,” is a list of frequently asked questions (FAQs) and answers about RapidApp operation.
- Appendix D, “Source Code for the Calculator Application,” shows the source code for the calculator application developed in Chapter 1 and throughout the book.

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 you 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 SGL_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 Intrinsic Programming Manual* describes how to write X Window System™ programs using the Xt Intrinsic 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 Intrinsic 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 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>.

RapidApp Tour

This chapter demonstrates basic RapidApp use by describing how to create a simple calculator application that adds two integers. Each step that you perform is described in detail in Chapter 3, Chapter 4, and Chapter 5.

Building a Calculator: A RapidApp Example

Figure 1-1 shows how the Calculator application looks when finished. Refer to Appendix D, “Source Code for the Calculator Application,” for the source code for this example.

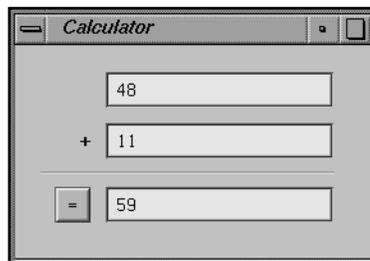


Figure 1-1 Completed Calculator Application

To create the calculator application:

1. Start RapidApp.

RapidApp’s main window opens as shown in Figure 1-2. The areas of this window are described in “The RapidApp Interface” on page 17.

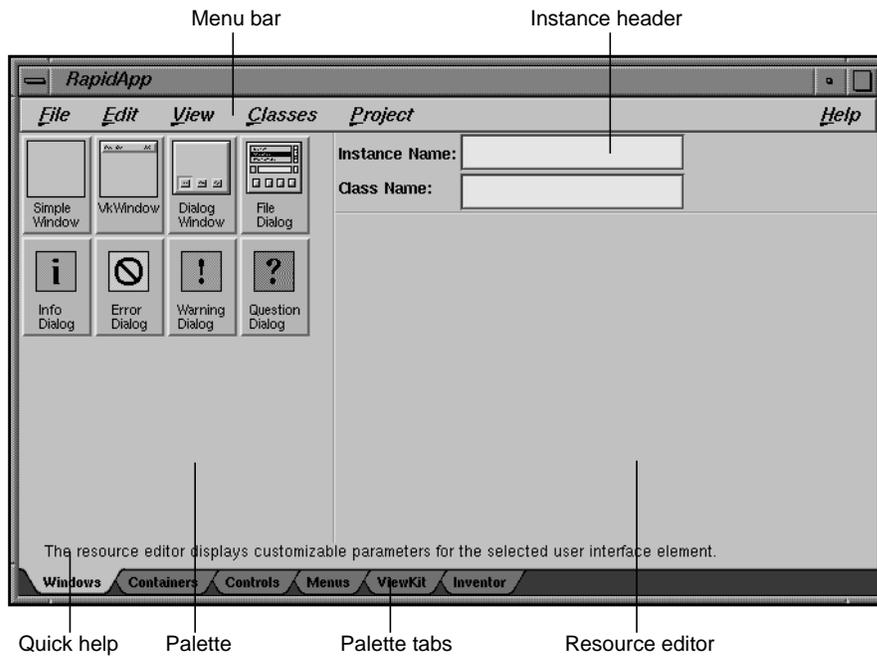


Figure 1-2 RapidApp’s Main Window

2. Create a top-level window.
 - From the Window palette, click the Simple Window icon.
 - Position the pointer somewhere on the screen and click the left mouse button to place the window.
 - In the Instance Name text field in the instance header area of the RapidApp’s main window, type “calcWindow” and press **<Return>**.
 - In the resource editor, find the resource named **title**, type “Calculator”, and press **<Return>**.

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

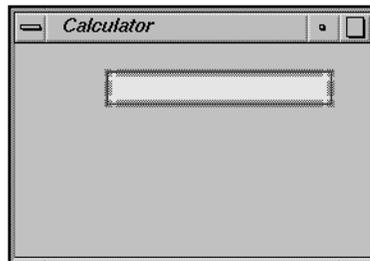


Figure 1-3 Initial Calculator Layout

5. Add a second Text Field below the first and place a label to the left of the second Text Field. Figure 1-4 shows the resulting layout.

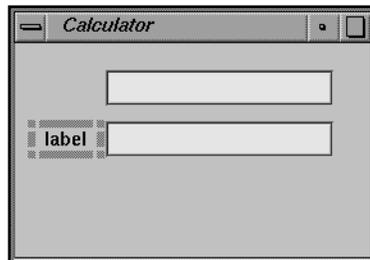


Figure 1-4 Second Text Field and Label

6. Complete the layout by adding a separator below the second Text Field, and a PushButton and third Text Field below the separator. Figure 1-5 shows the layout after all widgets have been placed.

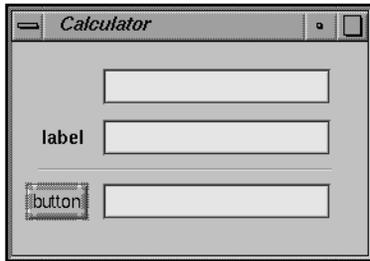


Figure 1-5 All Widgets In Place

7. Rename the top text field element "value1," the second "value2," and the third "result."
 - Click each text field element in turn
 - In the Instance Name field in the header area, enter the new name.
8. Change the label on the Label element 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 element 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".
 - Press **<Return>**.

Notice that RapidApp automatically adds "(" after the function name. At this point, the interface should look like the window in Figure 1-1.

11. Test the interface.

- From the View menu, choose “Play Mode.”
- 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 1-6), reporting that the **add()** callback is being called.

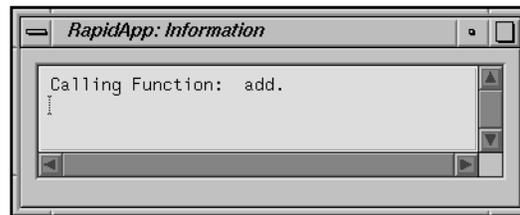


Figure 1-6 RapidApp Information Window

12. Set code generation options.
 - From the View Menu, choose “Build Mode” to go back to build mode.
 - From the File menu, choose “Preferences.”
 - In the Preferences dialog, go to the Project card.
 - Change the directory path to the directory where you want RapidApp to save the files for the application. If this directory doesn't exist, RapidApp asks if it should create the directory.
 - Change the name field to “calculator,” as shown in Figure 1-7. Make sure that the rest of the options are set as shown in Figure 1-7.
 - Click the *Ok* button.

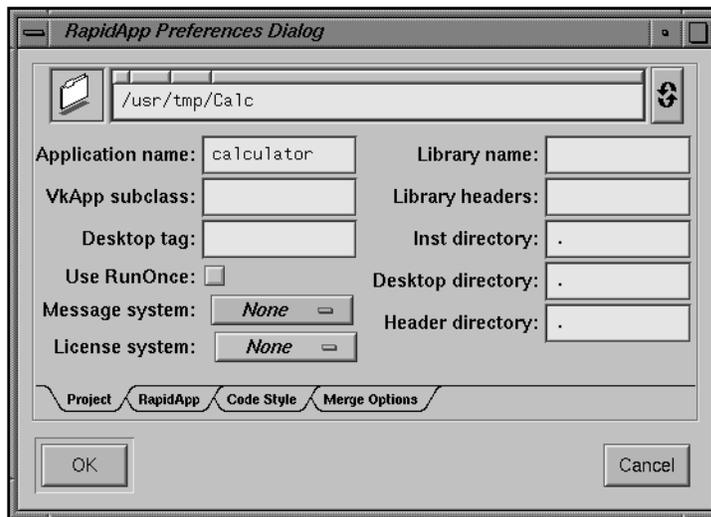


Figure 1-7 Project Card

13. Save the interface

- From the File menu, choose “Save.”

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 choosing “Generate C++” from the Project menu.

RapidApp displays a status window to report the files that it creates.

15. When the dialog displays indicating that the window was not declared as a class, click the *Continue, Don't Ask Again* button.

16. Build and run the application by choosing “Run Application” from the Project menu.

The Developer Magic Build Manager (see Figure 1-8) appears and compiles the application. Once compiled, the application runs automatically.

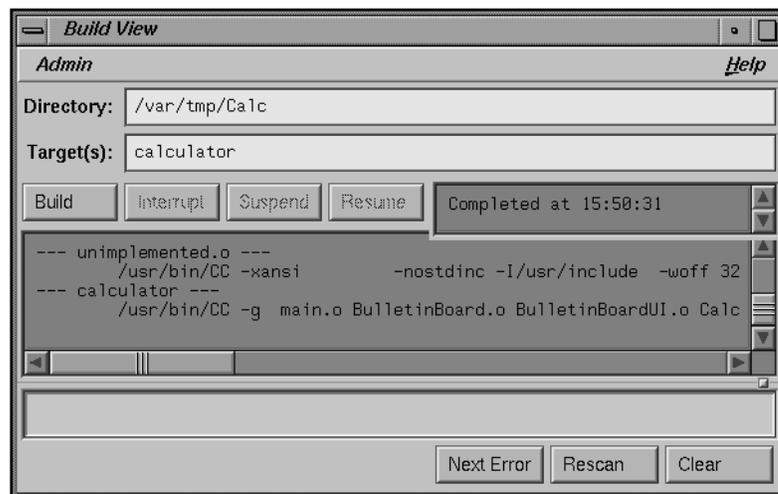


Figure 1-8 Building the Calculator Application

17. Add functionality.

- From the Project menu, choose “Edit File.”
- In the file selection dialog, choose the file *BulletinBoard.C*.
- When the text editor appears, scroll down until you locate the section of code shown in Example 1-1, which is the callback invoked when the user clicks the “=” pushbutton:

Example 1-1 Sample of Default Code Generated for Pushbutton Callback

```
void BulletinBoard::add ( Widget w, XtPointer callData )
{
    //---- Start editable code block: BulletinBoard add

    XmPushButtonCallbackStruct *cbs = (XmPushButtonCallbackStruct*) callData;

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

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

    //---- End editable code block: BulletinBoard add
} // End BulletinBoard::add()
```

- Edit this function so that it appears as shown in Example 1-2 (your additions are shown in **bold**).

Example 1-2 Example of Adding Functional Code to a Pushbutton Callback

```
void BulletinBoard::add ( Widget w, XtPointer callData )
{
    //---- Start editable code block: BulletinBoard add

    XmPushButtonCallbackStruct *cbs = (XmPushButtonCallbackStruct*) callData;

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

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

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

    //---- End editable code block: BulletinBoard add
} // 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 near the top of the file and find the line:


```
//---- Begin editable code block: headers and declarations
```

 After that line, add the header file for the **VkFormat()** function and `<stdlib.h>` for the **atoi()** function:


```
#include <Vk/VkFormat.h>
#include <stdlib.h>
```
- Save the file and close the editor.

18. Test the completed application.

- From the Project menu, choose “Run Application.”

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 application runs automatically. Figure 1-9 shows the completed application as it appears on the screen.

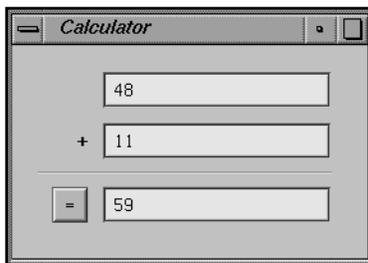


Figure 1-9 Working Calculator Application

- Try typing integer values into the text fields and pressing the “=” button.

19. Package the product so that other users can install the calculator application using the Software Manager software installation tool (*swmgr*). Note that you must have **inst_dev** installed.

- Go to the *Calc* directory and enter:

```
% make image
```

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

RapidApp Basics

This chapter provides an overview to RapidApp and useful information for using RapidApp effectively. It contains:

- “RapidApp Overview,” an overview of RapidApp
- “Prerequisites for Installing RapidApp,” a list of products you must have installed on your system to use certain features of RapidApp
- “Setting Up Your Working Environment for RapidApp,” a list of environment variables you might want to set for easier use of RapidApp
- “Starting RapidApp,” instructions for running RapidApp
- “The RapidApp Interface,” an overview of the RapidApp interface

RapidApp Overview

RapidApp is an interactive tool for creating applications. It integrates with other Developer Magic tools, including `cvd`, `cvstatic`, `cvbuild`, 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 xxvii.

Note: RapidApp is a tool that provides an easy-to-use interface between you and the SGI development tools and libraries such as the Debugger, IRIS IM, and ViewKit. It allows you to build your interface in the SGI style without requiring a full knowledge of SGI's development environment (though basic knowledge is recommended). As such, RapidApp does not dictate the behavior of these tools and libraries. If you encounter unexpected behavior, first check RapidApp's guide, and then check the guide appropriate for the tool or library with which you're working.

Prerequisites for Installing 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 which include the Developer Magic tools, such as the Static Analyzer (*cvstatic*) and the Build Analyzer (*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 before you install and run RapidApp.

Setting Up Your Working Environment for RapidApp

The following sections describe how to prepare your working environment for RapidApp.

Project Directory

SGI recommends that you set up a project directory for each application you build. When you build your application, RapidApp generates and saves a number of files automatically. Unless you specify a project directory, RapidApp saves these files in the current directory. To specify a project directory for RapidApp:

1. From the File menu, choose "Preferences."
2. Go to the Project card.
3. In the Project Directory field, enter the path to your project directory.
If the directory doesn't exist, RapidApp asks if it should create the directory.
4. In the Application name field, enter the name of your application.

Resource Files

When you create an application with RapidApp, it generates a resource file in the project directory containing all the application's resources (for example, labelString, a resource used by labels and push buttons). RapidApp automatically loads these resources when it runs your application, but your application might not find them if you try to run it independently from the command line.

To ensure that your application finds the resource file, set the environment variable `XUSERFILESEARCHPATH` to `"%N%S"` to add the current directory to the application's resource search path. You might want to do this as part of your login setup.

Editor

You can use any window-based editor within RapidApp. To do so:

1. Set the `WINEDITOR` environment variable to the window-based editor you want to invoke. For example:

```
setenv WINEDITOR 'winterm -c vi'
```
2. In RapidApp, from the File menu, choose "Preferences."
3. In the RapidApp card, set the "Use \$WINEDITOR" option.

If you do not specify your own editor, RapidApp uses the Source View editor.

Color-Map

By default, RapidApp installs its own color map. For machines with limited colormap support (for example, IRIS IndigoTM workstations), you can force RapidApp to use the default colormap by adding the following line to your `~/.Xdefaults` file:

```
RapidApp*useDefaultColormap: True
```

Starting RapidApp

To start RapidApp:

- In a shell window, type:

```
% rapidapp
```

–or–

- Use the RapidApp icon:
 - Go to the Toolchest, and from the Find menu, choose “An Icon.”
 - Search for “rapidapp.”
 - Drag the RapidApp icon to your desktop.
 - Start RapidApp by double-clicking on the icon.

The Startup Screen

When you launch RapidApp, it displays a startup screen as shown in Figure 2-1.



Figure 2-1 RapidApp’s Startup Screen

By default, you dismiss the startup screen once RapidApp's main window appears. You can control this behavior through the RapidApp Preferences dialog.

1. In the RapidApp main window, from the File menu, choose "Preferences."
2. In the Preferences dialog, go to the RapidApp card.
3. Set the "Auto-dismiss start screen" option.

The next time you launch RapidApp and once the main window appears, RapidApp will dismiss the startup screen automatically.

RapidApp Tips

By default, RapidApp's startup screen contains a "tip," a suggestion for how to use RapidApp. Figure 2-1 above shows the startup screen with a tip.

You can control whether or not the startup screen displays tips through RapidApp's Preferences dialog.

1. In the RapidApp main window, from the File menu, choose "Preferences."
2. In the Preferences dialog, go to the RapidApp card.
3. Unset the "Show tips on startup" option.

The next time you launch RapidApp the startup screen will not include a tip.

To view the complete list of tips see "RapidApp Tips" on page 271.

The RapidApp Interface

Figure 2-2 shows the main areas of the RapidApp main window. The following sections describe each of these areas.

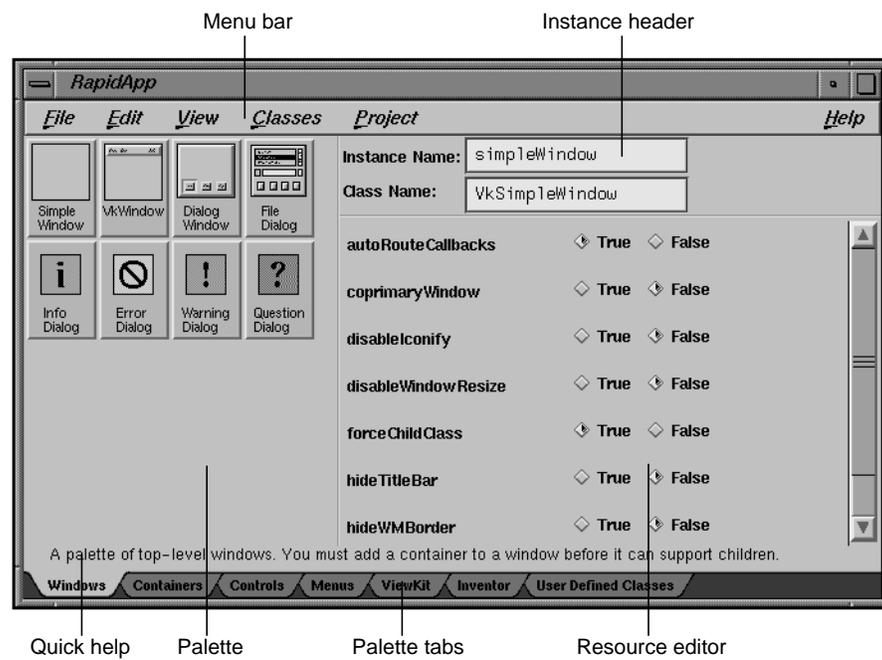


Figure 2-2 RapidApp's Main Window

Menu Bar

The RapidApp menu bar provides the following menus:

File	Allows you to create new RapidApp files, save and open RapidApp files, set RapidApp preferences and quit RapidApp.
Edit	Allows you to cut, copy, paste, and manipulate a selected interface element.
View	Allows you to switch 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 classes.
Project	Allows you to generate code, browse and edit files, build an application, run the program under a debugger, and so on.
Help	Allows you to access the online help system.

Palette

The palette on the left side of RapidApp’s main window contains icons representing interface components used to construct your interface. You access different palettes via tabs that appear along the lower portion of the window. By default, RapidApp contains the following palettes:

- Windows—interface elements that serve as top-level windows
- Containers—interface elements that serve as containers in which to group other interface elements
- Controls—basic interface elements for user interaction
- Menus—interface elements used to construct menus
- ViewKit— interface elements from the ViewKit library
- Inventor—interface elements from the Inventor library

Each of these palettes and their icons are described in detail in Appendix A, “RapidApp Reference.” Also, depending on the libraries you have installed, additional palettes and icons may exist.

Instance Header

The instance header displays the instance and class names of the currently selected interface element.

When you create an element, RapidApp automatically generates its instance name. You can change this by entering a new string in the Instance Name field. RapidApp uses this name when it generates code.

The class name is the widget class name 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 from `XmLabel` to `XmPushButton`.

Resource Editor

The resource editor on the right side of RapidApp's main window lists the customizable *resources* for a newly created or currently selected interface element. The contents of this area change dynamically, depending on the interface element selected.

Quick Help

The quick help area is immediately above the palette tabs. When you move the cursor over an item in RapidApp's main window or over an item in your interface, the quick help area displays a one-line message describing the item.

Palette Tabs

The palette tabs along the lower portion of RapidApp's main window provide access to different palettes. Click on a tab to access its palette.

Interface Design

This chapter provides the basic steps to designing an interface with RapidApp and techniques for working with interface elements. It contains the following sections:

- “The Indigo Magic Style”
- “Basic Interface Construction”
- “Parent/Child Interaction”
- “Interface Element Creation”
- “Interface Element Manipulation”
- “Save Your Interface and Files”

Also, the Calculator tutorial in Chapter 1 steps you through the basic use of RapidApp as does the “Getting Started” tutorial available through the Help menu in RapidApp’s main window.

The Indigo Magic Style

One of RapidApp’s features is to automatically provide the Indigo Magic style when you create interface elements using RapidApp. To learn more about this style, 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*.

Basic Interface Construction

Before you begin building the interface to your application, you should understand the basic steps for constructing an interface using RapidApp. If you are using RapidApp to construct a component library, see Chapter 8.

1. From the Windows palette, create a top-level window.
Top-level window's can contain exactly one interface element.
2. From the Containers palette, choose and add a container to the window.
Containers can contain one or more interface elements (including other containers).
3. From any palette (except the Windows palette), choose and add one or more interface elements to the container.

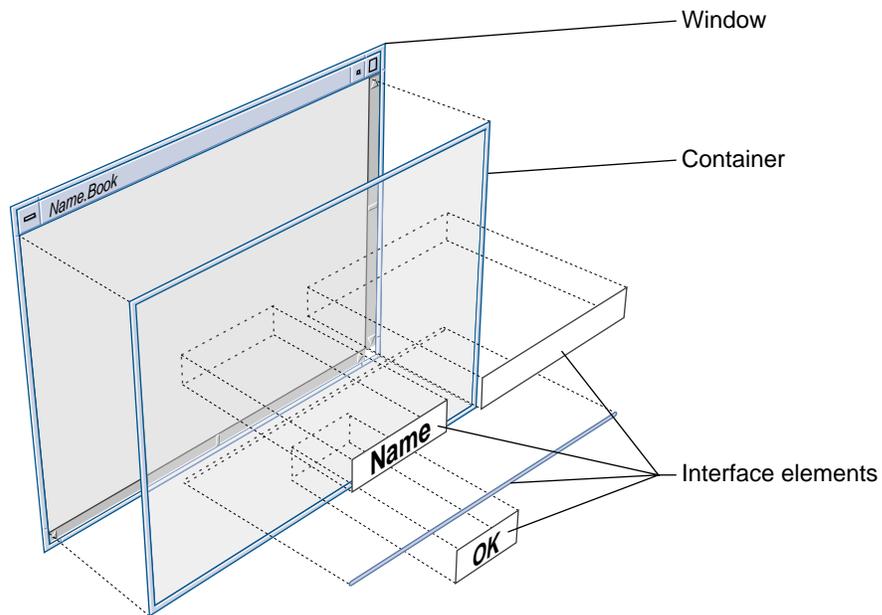


Figure 3-1 Constructing an Interface

See Also

- “Work With Windows”
- “Work With Containers”
- “Parent/Child Interaction”
- “Interface Element Creation”

Parent/Child Interaction

This section describes the following:

- “Parent/Child Relationship”
- “Selecting a Parent Element”
- “Confining Element Creation to the Selected Parent”

Parent/Child Relationship

When an interface element is added to another interface element, it becomes a child of that element. A container is a child of a window (or another container), and any element added to a container is a child of that container. Figure 3-2 diagrams the parent/child relationship.

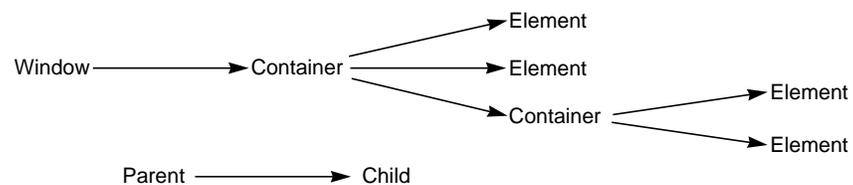


Figure 3-2 Parent/Child Relationship

Selecting a Parent Element

To select an element's parent, you click on the parent. When you can't determine the parent, or can't click on the parent (e.g., you can't click on a shell widget or a window that has a child), do the following:

1. Select the child element.
2. Choose "Select Parent" from the Edit menu in RapidApp's main window.

RapidApp selects the element's parent.

Confining Element Creation to the Selected Parent

By default, when you create an element by dropping it onto a valid parent, the new element becomes a child of that parent even if you have another parent selected. This behavior is a result of pointer focus mode.

If you want to ensure that elements are added to the selected parent only, change to explicit focus mode by choosing "Keep Parent" from the View menu. In this mode, any element you create is added to the currently selected container, no matter where you drop the child in the application you're creating or in the RapidApp interface.

As a further convenience, when "Keep Parent" is set, RapidApp grays out icons that you can't add to the currently selected container. For example, if you select a menu bar when "Keep Parent" is set, RapidApp grays out all menu palette icons except those you can add to the menu bar.

Interface Element Creation

The section describes the following:

- "Creating Interface Elements"
- "Copying Interface Elements"
- "Aborting Interface Element Creation"
- "See Also"—a reference to additional information

Creating Interface Elements

This section steps you through the process of creating an interface element. To create an interface element:

1. Decide what you are creating.

You need a window to which you add a container to which you add your remaining interface elements.

Or, you need a window to which you add a complex component. Complex components are described in section “Work With User-Defined Components” on page 104.

If you are creating a menu or dialog, see sections “Work With Menus” on page 92 and “Work With Dialogs” on page 97.

2. Go to the appropriate palette by clicking on its tab.
3. Click the desired icon.

An outline appears, representing the default size of the element.

Note: RapidApp enforces a minimum size of 20x20 pixels for all interface elements, and the window manager enforces minimum sizes for its direct children (shell widgets). For example, in certain containers, you can resize a button down to 20x20 pixels. However, if a button has its own shell, its size is determined by the minimum size allowed for the shell. 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.

4. If the element is a window, move the outline over the desktop; otherwise, move the outline over your top-level window.
5. Press the left mouse button to position the upper left corner of the element.
6. Release the mouse button to accept the element’s default size.

–or–

Drag out a new size before releasing the button.

Unless added to another container, container elements automatically fill a window’s entire work area.

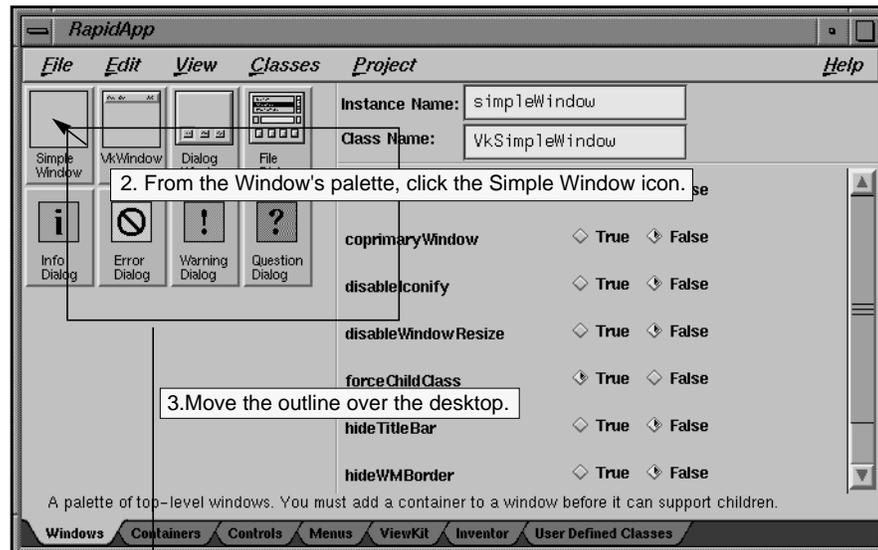
To create an interface element, other than a window, using the drag and drop method:

1. With the cursor over an icon, press the middle mouse button.
2. Drag the cursor over a window or container and release the mouse button.

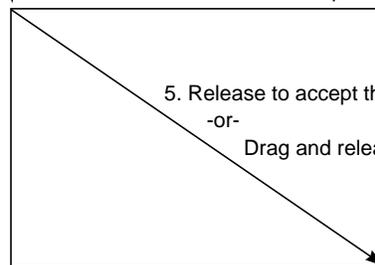
The element displays in its default size.

Figure 3-3 demonstrates how to create a Simple Window.

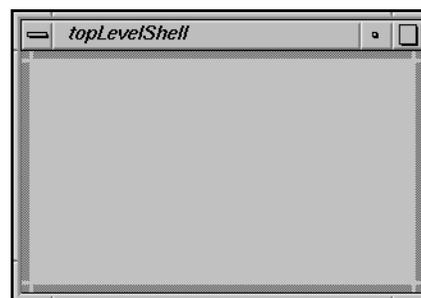
1. Decide what you are creating (in this example, a Simple Window).



4. Press the left mouse button to position the upper left corner.



5. Release to accept the default size.
-or-
Drag and release to change the size.



The window appears.

Figure 3-3 Creating a Simple Window

Copying Interface Elements

When your interface requires several instantiations of the same interface element—same size, same label, same border thickness, etc.—you can simplify creation by designing one element and then copying it. A copied element retains the settings of the original except for the instance name which must be unique. RapidApp uniquely names each copied element by appending a number to the original name (e.g. copied versions of “myButton” are named “myButton1,” “myButton2,” etc.).

To copy an interface element:

1. Select the element.
2. From the Edit menu, choose “Copy.”
3. From the Edit menu, choose “Paste.”

An outline of the element appears

4. Place the new element by clicking in the target container.

Aborting Interface Element Creation

To abort the creation of an interface element after selecting an icon, click the middle button.

See Also

- “Basic Interface Construction”
- “Work With Windows”
- “Work With Containers”
- “Work With Menus”
- “Work With Dialogs”
- “Building a Calculator: A RapidApp Example”
- The “Getting Started” tutorial accessed by choosing “Tutorial>Getting Started” from the Help menu in RapidApp’s main window.

Interface Element Manipulation

This section describes the following:

- “Moving and Resizing Interface Elements”
- “Deleting Interface Elements”
- “Naming Interface Elements”
- “Modifying an Interface Element’s Appearance or Behavior”
- “Additional Interaction Techniques”

Moving and Resizing Interface Elements

How you move or resize an element in RapidApp depends on the type of element you’re working with and the type of its parent element. The following sections are included:

- “Moving and Resizing Windows”
- “Moving and Resizing Containers”
- “Moving and Resizing Elements within Containers”
- “Moving Elements to Another Container”

Note: RapidApp enforces a minimum size of 20x20 pixels for all interface elements, and the window manager enforces minimum sizes for its direct children (shell widgets). For example, in certain containers, you can resize a button down to 20x20 pixels. However, if a button has its own shell, its size is determined by the minimum size allowed for the shell. 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.

Moving and Resizing Windows

You move and resize windows just as you would any desktop window.

Moving and Resizing Containers

A container with a window as its parent, maintains the same size and position as the window’s work area.

A container with another container as its parent, depends on its parent when being resized or moved. To understand how parent containers control their child elements, see “Moving and Resizing Elements within Containers” below.

Tip: If your interface requires containers within containers, consider constructing the inner container in its own window. Resize and move its child elements as necessary. When complete, move the container and its children into the other container, or create a user-defined component comprised of the elements and instantiate this component into the other container. For more information on user-defined components, see “Work With User-Defined Components” on page 104.

Moving and Resizing Elements within Containers

A container controls the position and size of its child elements. To help you understand how to work with elements in their containers, the following sections are included:

- “Introduction”
- “Using the Grid”
- “Directly Moving Child Elements” of flexible containers
- “Directly Resizing Child Elements” of flexible containers
- “Indirectly Moving Child Elements” of controlling containers

Introduction

RapidApp uses IRIS IM container widgets, a collection of containers that provide a variety of ways to organize interface elements. To maintain their organizational features, many containers keep a tight control over their child elements, often determining their size and position by the order in which the elements were created.

For example, the Row Column container supports menu bars and menu panes. For this reason, it forces all of its children to have the same height and limits their ability to be resized. The Bulletin Board container is a simple more flexible container. As such, it allows its children to be resized and moved without limitation (except to remain within the container’s boundary).

Note: How these containers behave is a function of Xt and IRIS IM, and not of RapidApp.

To understand how to manipulate child elements, you must understand and choose the appropriate container in which to place them. For information on the containers used by RapidApp, see “Work With Containers” on page 72.

Using the Grid

RapidApp provides a grid to help you control the placement and size of interface elements. As you move or resize an element, it snaps to the grid. To control the grid:

1. From the View menu, choose “Snap to Grid.”
2. Set the resolution to 2, 5, 10, or 20 pixels

–or–

Turn the grid off.

Directly Moving Child Elements

You can easily move child elements in the following containers:

- Bulletin Board
- Rubber Board
- Form (elements attached to other elements influence each others size and position)

Note: Some interface elements are actually components and are comprised of several interface elements including a container. For example, the Option Menu available on the Menus palette is comprised of a RowColumn container with a menu element floating inside. These elements are not easily resized, and in some cases cannot be resized.

To move a child element using the cursor:

1. With the cursor over the element, press the left mouse button.
2. Drag the element to its new position.
3. Release the mouse button.

Figure 3-4 illustrates this process.

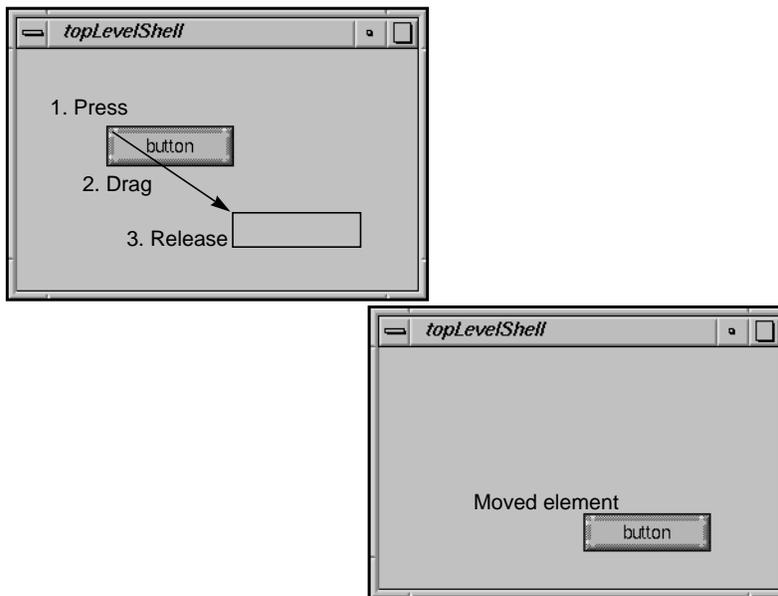


Figure 3-4 Moving an Interface Element

To move a child element using the arrow keys:

1. Select the element.
2. Press the appropriate arrow key.

The element moves one pixel ignoring the grid setting.

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.

Directly Resizing Child Elements

You can easily resize child elements in the following containers:

- Bulletin Board
- Rubber Board
- Form (elements attached to other elements influence each others size and position)

To resize a child element

1. Select the element.
2. With the cursor over one of the handles surrounding the element, press the left mouse button.
3. Drag the handle until the element is the desired size.
4. Release the mouse button.

Figure 3-5 illustrates this process.

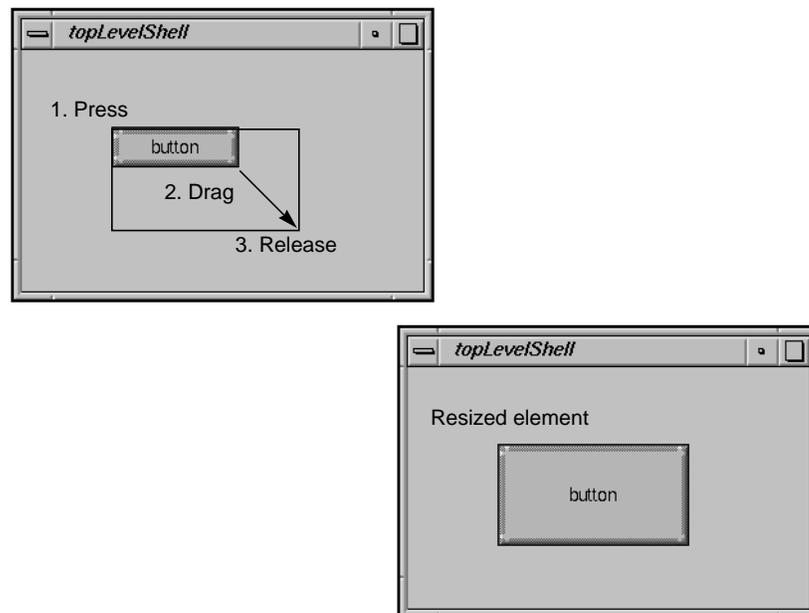


Figure 3-5 Resizing an Interface Element

Hint: Some elements have a **recomputeSize** resource. When set to true, the element resizes to accommodate its label string when the label string is changed. If you set the size of such an element and then change its label string, the element will resize. To prohibit this behavior, set the **recomputeSize** resource to false. For more information on resources, see “Modifying an Interface Element’s Appearance or Behavior” on page 37.

If the element is too small to resize:

1. Select the element.
2. From the Edit menu, choose “Grow Widget.”

The width and height of the selected element increases by 20 pixels.

To return an element to its default size:

1. Select the element.
2. From the Edit menu, choose “Natural Size.”

Note: The “Natural Size” option has no effect if the element is a child of a container that controls its size (for example, a Row Column).

Indirectly Moving Child Elements

For containers in which a child’s position depends on the order in which it was created, changing the child’s x,y position is ignored. In many cases, though not all, resizing a child of such a container is also ignored. Though RapidApp cannot control this behavior, it can create the illusion of movement through the use of “Up/Left” and “Down/Right” on the Edit menu or through the use of arrow keys. When you use one of these commands or one of the arrow keys, the selected element appears to move in the appropriate direction. RapidApp is actually altering the element’s creation order. Figure 3-6 shows an example of repositioning a toggle button in a row column container.

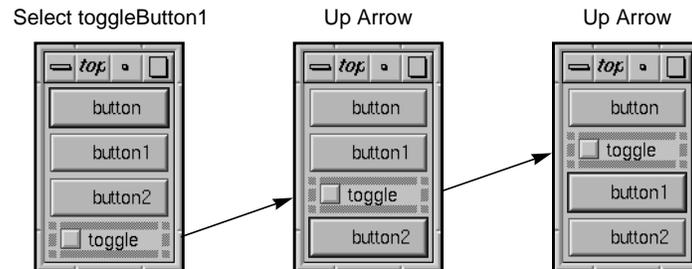


Figure 3-6 Moving an Interface Element in a RowColumn

Moving Elements to Another Container

In addition to moving an element within a container, you can move an element to another container.

To move a child element via the Edit menu:

1. Select the element.
2. From the Edit menu, choose “Cut” or “Copy.”
3. From the Edit menu, choose “Paste.”

An outline of the element appears.

4. Place the element by clicking in the target container.

When you copy an interface element, all of its settings are copied with it, except for the instance name which must be unique. RapidApp creates a unique name by appending a number to the name (e.g. a copy of “myButton” is named “myButton1”).

To move a child element using the middle mouse button:

1. With the cursor over the element, press the middle mouse button.
2. Drag the element to the target container.

If you hold down the <ctrl> key while dragging, RapidApp copies the selected element instead of moving it.

Deleting Interface Elements

To delete an element:

1. Select the element.
2. From the Edit menu, choose "Cut." This option saves the element to the clipboard.

–or–

From the Edit menu, choose "Delete." (or press <Backspace> or <Delete>). This option doesn't save the element to the clipboard.

There is no undo feature.

Naming Interface Elements

All interface element names must be unique. When you create an interface element, RapidApp assigns it a unique name automatically. For example, RapidApp gives the name "button" to the first push button you create, "button1" to the next button you create, and so on. The name determines both the string given to the element (its resource name) when it is created, and the variable that represents the element in the program.

To change an element's name:

1. Select the element.
2. Edit the Instance Name field in the header area of RapidApp's main window.

Figure 3-7 shows the header area with a user-specified name displayed in the Instance Name field.

Instance Name	bigRedButton
Class Name	XmPushButton

Figure 3-7 Changing an Element's Name in the Header Area

Modifying an Interface Element's Appearance or Behavior

You control an interface element's appearance and behavior by setting the element's *resources*. For example, to change the string of a push button, you change its `labelString` resource. The following sections are included:

- "Modifying Resources"
- "Callback Resources"
- "Constraint Resources"
- "Dynamic Resources"
- "Extended Resources"

Modifying Resources

To modify an element's resources:

1. Select the element.

The resource editor displays the element's resources.

2. Change the desired resource as follows:
 - For string values – edit the text field. The value is accepted when you press **<Return>** or click outside of the field.
 - For Boolean values – click the desired toggle.
 - For enumerated values – choose a value from the option menu, by pressing on the option button to the right of the resource.

Figure 3-8 shows the resources for a push button.

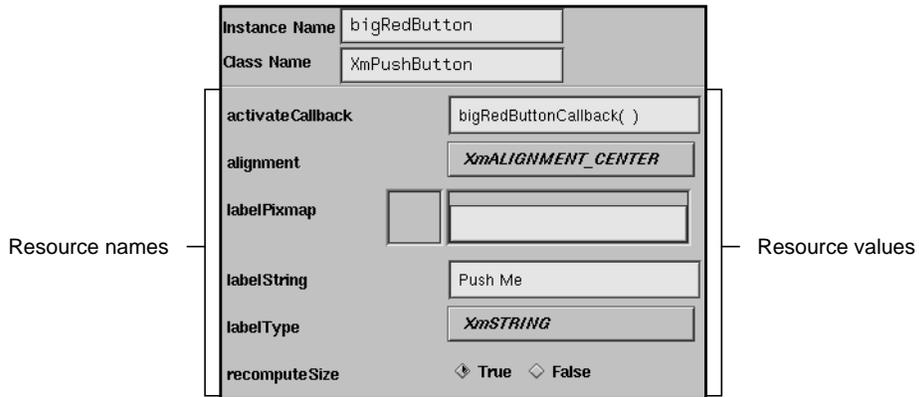


Figure 3-8 Push Button Resources

IRIS IM widgets are highly configurable and often include a large number of resources. Typically, when writing an application, you need to access only a few, so RapidApp displays the most commonly used resources. You can access all resources programmatically by editing the source code generated by RapidApp or by editing the application’s resource file. Colors and fonts are controlled by schemes in the Silicon Graphics environment. Colors can be changed using the Color Scheme editor available as part of the Indigo Magic desktop.

Callback Resources

Callbacks are functions that associate program behavior with user input. For example, a push button interface element has an callback function that is activated when the user clicks the button. To specify this function, you enter the function’s name in the push button’s **activateCallback** resource, as shown in Figure 3-9. (You don’t have to enter the parentheses; RapidApp automatically provides them when you finish editing the 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” in Chapter 5 for more information on editing generated code to implement functionality.)



Figure 3-9 Adding a Callback

Constraint Resources

IRIS IM (and, therefore, RapidApp) supports the concept of constraints—resources added to an element when it is a child of a particular type of container. Constraint resources assist the container in controlling the appearance and behavior of its child elements.

For example, a label element always has the following resources: **alignment**, **labelPixmap**, **labelString**, **labelType**, **recomputeSize**, and **schemeFont**. When added to a frame container, a container that typically has a work area with a title, the container adds the following constraint resources to the label: **childHorizontalAlignment**, **childType**, and **childVerticalAlignment**. This allows the container to control the position of the label after you indicate that the label is a title. This is the reason you may see elements of the same type with different resources.

The resource editor lists constraint resources separately from other resources, below a label identifying them as constraint resources. You modify constraint resources just as you do other resources. Figure 3-10 shows the constraint resources added to a label element when it is a child of a frame container.

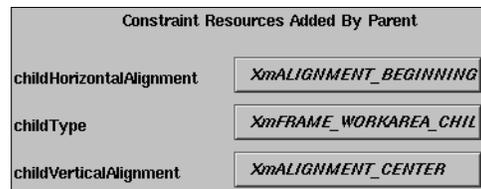


Figure 3-10 Constraint Resources

Dynamic Resources

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

Also, RapidApp displays some resources only when appropriate. For example, the push button supports an accelerator resource that describes a key combination used to activate the button when it is in a menu. RapidApp displays this resource only when the button is in a menu; it is meaningless otherwise.

Extended Resources

Extended resources provide interface elements with additional behavior; behavior that isn't in Motif. For example, The Indigo Magic environment uses co-primary windows—auxiliary windows that aren't displayed on startup. The term “co-primary” is unknown to Motif. To enable you to create a co-primary window, RapidApp includes a `coprimaryWindow` extended resource for both the simple window and `VkWindow`.

Additional Interaction Techniques

This section describes some additional techniques for working with interface elements, and includes the following sections:

- “Locking on to an Element”
- “Viewing the Interface Element Hierarchy”

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 containers 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, hold down `<Ctrl>` while manipulating the selected interface element.

Viewing the Interface Element Hierarchy

To help you see the structure of the interface element hierarchy, You can have RapidApp color code the nested containers within the hierarchy.

To view an interface element hierarchy by color coding:

- Choose “Color by Depth” from the View menu.
RapidApp colors the containers differently within the same hierarchy.
This is a toggle command and choosing it again turns off the color coding.

Figure 3-11 shows a window containing three buttons. Figure 3-12 shows the same window with “Color by Depth” turned on. You can see the depth of this hierarchy by the fact that the nested containers are colored differently.

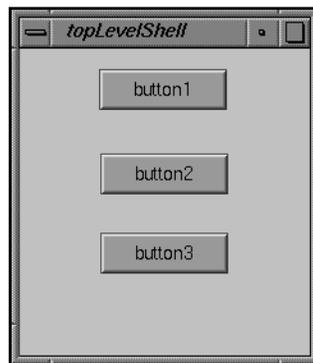
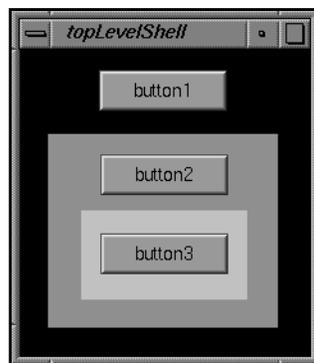


Figure 3-11 Window with Three Labels



3 Levels

- Container colored black
- Container colored blue
- Container colored gray

Figure 3-12 Window with Colored Interface Element Hierarchy

For a more detailed description of the hierarchy, use the *editres* program. For more information, see the *editres(1)* reference page.

Save Your Interface and Files

In RapidApp, you save a description of your application's interface using the "Save As" File menu option. You have RapidApp create and save all your application files using the "Generate C++" Project menu option. In both cases, you should first specify the project directory in which to save these files.

To specify the project directory:

1. From the File menu, choose "Preferences."
2. In the Preferences dialog, go to the Project card.
3. In the Project Directory field, enter the directory path for the project directory.
4. Fill in the other options as necessary.

To save a description of the interface:

1. From the File menu, choose "Save As."
2. In the Save As dialog, provide the project directory and the name of the file. The file name must end with a *.uil* suffix.

The *.uil* suffix is a file format used by IRIS IM, as well as by many user interface tools. If this suffix is missing, RapidApp adds it automatically.

Note: The "Window Template" option on the Save As option menu is discussed in "Adding New Top-Level Window Layouts to RapidApp" on page 70.

To have RapidApp create and save all application files:

- After you've built your interface, choose "Generate C++" from the Project menu.

For a list of generated files, see "Code Generation" on page 55.

Application Development

This chapter describes the application development cycle and the application development model best supported by RapidApp. It includes the following sections:

- “Development Cycle” describes the typical development cycle for creating an application with RapidApp.
- “Development Model” discusses the general development model supported by RapidApp.

Development Cycle

The RapidApp application development cycle typically consists of the following steps.

1. Use RapidApp to create a graphical user interface for your application.
See Chapter 3, “Interface Design.”
2. Before you build, debug, or run your application, generate the C++ code.
See “Generating RapidApp’s Code” on page 49.
3. Add user-defined code to your application.
See “Generate Code” on page 49.
4. Test your application.
See “Test Your Application” on page 44.
5. Use RapidApp to refine your interface based on testing and feedback.
6. Develop and test any external (that is, non-interface) functionality required by your application.
7. If you used EZ convenience functions, replace them with production code.
8. Perform final testing and make necessary revisions.

9. Create a custom Desktop icon for your application, and edit the FTR file to customize the icon's behavior.
See "Create an Application Icon" on page 45.
10. Create a minimized window icon to represent your application when iconified.
See "Create a Minimized Window Icon" on page 46.
11. Customize your application's installable images.
See "Create Installable Images" on page 46.

Test Your Application

Testing an application involves compiling, running, debugging, etc. All of these tasks are accessed through the Project menu. When activated, these tasks launch Developer Magic tools, such as the Build Manager and the Debugger, which must have been previously installed. For information on these tools, see the *Developer Magic: ProDev WorkShop Overview*.

For example, to compile your application, choose "Build Application" from the Project menu. This launches 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. From the File menu, choose "Preferences."
2. In the Preferences dialog, go to the RapidApp card.
3. Set the "Generate Cvstatic Database" option and close the dialog.
4. From the Project menu, choose "Generate C++."
This updates your files, including 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. To launch the Static Analyzer:

- After creating the static analysis files, choose “Browse Source” from the Project menu.

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

Create an Application Icon

Typically, applications that appear on an SGI workstation have a desktop icon which, when double-clicked, launches the application. RapidApp offers a generic desktop icon in the *icon.fti* file. To include an icon with your application, do the following:

1. Customize the generic desktop icon, using IconSmith.
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.
2. If necessary, modify the behavior of the icon by editing the *desktop.ftr* file.
3. If you plan to publish and distribute your application, supply a unique desktop tag.

The desktop tag lets the system know that the icon is tied to your application. (RapidApp supplies a default tag which could conflict with other applications developed through RapidApp.)

- To get a desktop tag, send email to desktoptags@sgi.com.
- When the tag is returned to you, go to the Project card in the Preferences dialog and enter the tag number in the Desktop tag field.

When RapidApp generates code, it places the tag in the *icon.ftr* file automatically.

Create a Minimized Window Icon

When the user iconifies your application, the icon should be representative of your application. RapidApp supplies a default icon in the `<Application>.icon` file where `<Application>` is the name of your application. This is an `sgi.rgb` image file and can be modified in any editor that handles such files.

For more information, 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.

Create Installable Images

RapidApp automatically generates the files required to create an image that users can install with the Software Manager installation tool (*swmgr*).

To generate a default image:

- Go to the directory that contains your source and enter:

```
% 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:

- From the Project menu, choose “Edit Installation.”

This 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.”

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.

The key features of RapidApp’s development model, which are described in following sections, are:

- “Object-Oriented Components”—describes how RapidApp builds applications from object-oriented components
- “Code Management”—describes how RapidApp manages automatically generated and user-generated code.

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 one or more widgets. 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” 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 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.

Code Creation and Management

This chapter describes the steps for generating code and RapidApp's approach to code management. It includes the following sections:

- "Generate Code" describes how you generate code.
- "Code Management" discusses how RapidApp manages the code it generates and the code you write.

Generate Code

The following sections are included:

- "Generating RapidApp's Code"
- "Adding and Editing Code"

Generating RapidApp's Code

RapidApp generates and saves all the files needed for your application when you choose "Generate C++" from the Project menu. You need to generate code anytime you change the interface to your application, such as adding a new interface element or moving an interface element, or when you change any of the options in the Preferences dialog.

To have RapidApp generate code:

1. If you haven't done so, specify the project directory as follows:
 - From the File menu, choose "Preferences."
 - In the Preferences dialog, go to the Project card.
 - In the Project Directory field, enter the directory path for the project directory.
 - Fill in the other options as necessary.
2. From the Project menu, choose "Generate C++."

RapidApp creates all the files needed for your application and saves them in the specified project directory. For information on how RapidApp handles the code it generates and the code you write, and for a list of the files RapidApp generates, see “Code Management” on page 55.

Adding and Editing Code

You can add and edit code using one or a combination of two methods; the a text editor, such as Source View, and the Debugger using the Fix and Continue tool. To generate a working prototype quickly, you might want to use the VkeZ convenience functions. This section includes:

- “Following a Few Simple Coding Rules”
- “Using Colors and Fonts”
- “Using the Debugger to Add Code”
- “Using a Text Editor to Add Code”
- “Using EZ Convenience Functions”

Following a Few Simple Coding Rules

RapidApp uses a code merging process to bring the generated code and the user-defined code together. This merging process is discussed in section “Code Merging.” To guarantee that the code is merged properly, follow a few simple rules:

- Do not arbitrarily reformat any generated files.
- Try to limit your changes to the areas between the editable code blocks:

```
//---- Start editable code block:  
  
//---- End editable code block:
```

RapidApp owns the areas outside of these code blocks. You can make changes to these outside areas, but doing so increases the chances of your code conflicting with code generated by RapidApp when making later revisions to your application.

Tip: If you are using Emacs version 19, refer to RapidApp’s release notes for information on how to configure Emacs to highlight and confine your input to the editable code blocks.

- Use only RapidApp to make changes to the user interface.
Note: If you attempt to alter your application's interface by editing source code, RapidApp won't reflect your changes. This is because RapidApp doesn't read the source code but instead reads a separate file used to save a description of the interface. your changes aren't lost; RapidApp simply doesn't recognize them. For example, if you cut the creation of a widget directly from the interface source code, the widget still appears the next time you run RapidApp.
- Use the variables at the top of the *Makefile* to add files and set flags. These are:
 - *IMAGEDIR* — the location of inst images built by `make image`
 - *USERLIBS* — use to include addition libraries created outside of RapidApp
 - *OPTIMIZER* — use to set `-O` or `-g` flags for the compiler
 - *USERFLAGS* — use to include additional files created outside of RapidApp
 - *USERFLAGS* — use to set flags for the compiler

By working within RapidApp's intended model, you take advantage of RapidApp's best features in helping you create and build your application. If you work outside of this model, such as reordering functions, reformatting code outside of the editable code blocks, or modifying a file to the point that RapidApp can't identify the original structure programmatically, your ability to continue to use RapidApp is severely limited.

Using Colors and Fonts

RapidApp prevents you from setting the color or font of an interface element from within RapidApp by not displaying their resources. Instead RapidApp lets SGI's schemes mechanism assign the fonts and colors based on the user's selected scheme. You can still edit the source code or resource file to override the default colors or fonts, 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*.

Using the Debugger to Add Code

RapidApp provides access to the Developer Magic Debugger. Aside from the obvious uses for finding and fixing bugs, you can use the Debugger's Fix+Continue feature to interactively add functionality to your application. To learn more about this tool, see the *Developer Magic: Debugger User's Guide*. The steps in this section get you started with the Debugger and show you how to use its Fix+Continue feature.

Note: Though you access the Debugger through RapidApp, when you work with the Debugger you are actually working outside of RapidApp, and thus outside of RapidApp’s control.

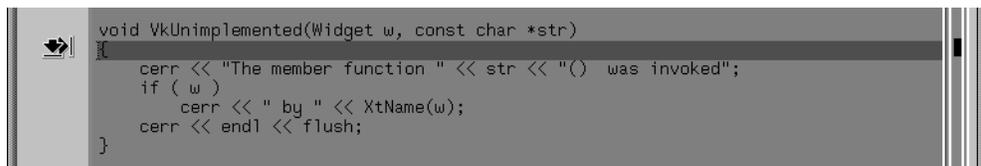
To use the Debugger from RapidApp:

1. From the Project Menu, choose “Debug Application.”

If the application is not up-to-date, RapidApp invoke the Build Manager automatically to update the executable.

2. Run your application by clicking on the *Run* button.
3. Test the application by clicking on buttons, selecting menu items, etc.

When you encounter an unimplemented function, the Debugger stops in **VkUnimplemented**. This function is called by the virtual function you really want to modify. Figure 5-1 shows an example of the **VkUnimplemented** function.



```
void VkUnimplemented(Widget w, const char *str)
{
    cerr << "The member function " << str << "() was invoked";
    if ( w )
        cerr << " by " << XtName(w);
    cerr << endl << flush;
}
```

Figure 5-1 VkUnimplemented Function

4. To access your function, click the *Return* button in the Debugger.

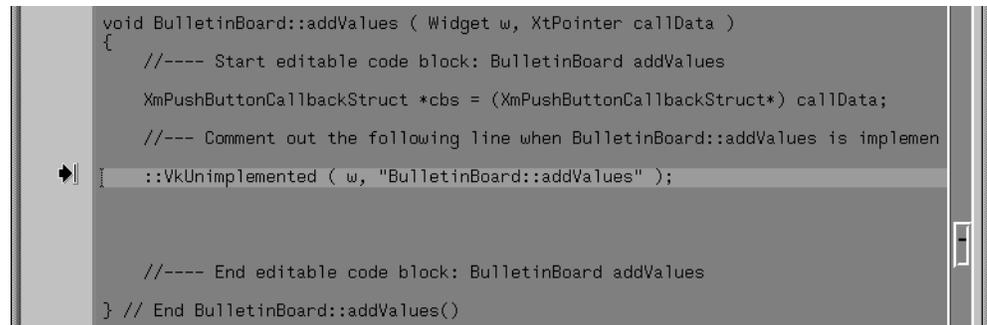
The Debugger displays the file containing the undefined function and highlights the undefined function. Figure 5-2 shows the **addValues** undefined function. Notice that the area of the function that you need to edit lies between the editable code blocks:

```
//---- Start editable code block

//----End editable code block
```

Notice also the line that reads:

```
::VkUnimplemented (w, "BulletinBoard::addValues" );
```



```
void BulletinBoard::addValues ( Widget w, XtPointer callData )
{
  //---- Start editable code block: BulletinBoard addValues
  XmPushButtonCallbackStruct *cbs = (XmPushButtonCallbackStruct*) callData;
  //--- Comment out the following line when BulletinBoard::addValues is implemen
  ::VkUnimplemented ( w, "BulletinBoard::addValues" );

  //---- End editable code block: BulletinBoard addValues
} // End BulletinBoard::addValues()
```

Figure 5-2 Example of an Undefined Callback Function

5. Define the function.
 - From the Fix+Continue menu, choose “Edit.”
The body of the function changes color to indicate the editable region.
 - Implement the behavior of the function.
6. Parse the modified function and load it for execution by choosing “Parse and Load” from the Fix+Continue menu.
7. Restart your application by clicking on the *Continue* button.
8. Continue to test your application making changes as necessary.
9. When you are satisfied with the testing, you might want to comment out the call to **VkUnimplemented** in all the functions you’ve changed.
If you don’t comment out this line, the function continues to be called.
10. Save your work by choosing “Save All Files” from the Fix+Continue menu.

Using a Text Editor to Add Code

You can use the default text editor, Source View, or any editor of your choice.

To set up the use of another editor:

1. Set the WINEDITOR environment variable to the window-based editor you want to invoke. For example:

```
setenv WINEDITOR 'winterm -c vi'
```

2. In RapidApp, from the File menu, choose "Preferences."
3. In the RapidApp card, set the "Use \$WINEDITOR" option.

To use the Source View editor or an editor previously specified as shown above:

1. From the Project menu, choose "Edit Files."
2. Make your changes trying to limit them to the editable code blocks:

```
//---- Start editable code block  
  
//----End editable code block
```

3. Save your changes through the Save feature of the chosen editor.

Using EZ Convenience Functions

During the prototyping stage, you can use EZ convenience functions in your callbacks. Specific to RapidApp, EZ functions help you write code to perform operations that would normally require significant knowledge of how IRIS IM deals with reconfiguring widgets, retrieving values, displaying data, etc. For more information on EZ functions, see Chapter 7, "VkeZ Library."

Code Management

RapidApp generates all the files needed to build your application. Although RapidApp can generate a significant portion of your application's code, you must still write some parts. 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: class architecture and code merging. This section describes how RapidApp manages code and includes the following sections:

- "Code Generation"
- "Class Architecture"
- "Code Merging"

Code Generation

To understand how all of the files that RapidApp generates work together, consider a very simple example—the calculator application described in "Building a Calculator: A RapidApp Example." To demonstrate more of the RapidApp code generation features, assume that you've encapsulated the calculator interface into a component called **Calculator** ("Work With User-Defined Components" demonstrates how to do this).

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

In the case of the calculator program, RapidApp generates the following directories and files:

Table 5-1 shows the files generated by RapidApp.

Table 5-1 Files Created by RapidApp

File/Directory Name	Description	Category
<i>.backup</i> (dir)	A directory containing backups of previous versions of files.	Support
<i>.buildersource</i> (dir)	A directory supporting RapidApp code merge features.	Support

Table 5-1 (continued) Files Created by RapidApp

File/Directory Name	Description	Category
<i>main.C</i>	<p>All programs have a <i>main.C</i> file that instantiates a VkApp object and one or more top-level windows. Alternatively, if you provide a subclass name in the RapidApp Preferences dialog, RapidApp creates a subclass of VkApp for the application instead of directly instantiating a VkApp object. This allows you to modify the VkApp subclass to handle application-specific needs (for example, parsing command-line options). See Chapter 3, “The ViewKit Application Class,” in the <i>IRIS ViewKit Programmer’s Guide</i> for more information on the VkApp class. You don’t have to use the generated <i>main.C</i>, but there is little reason to have anything different.</p> <p>Note: If you want to handle ToolTalk™ messages in your application, you need to set the Message System option in the RapidApp Preferences dialog to ToolTalk. This causes RapidApp to instantiate a VkMsgApp object instead of a VkApp object. See Appendix A, “ViewKit Interprocess Message Facility,” in the <i>IRIS ViewKit Programmer’s Guide</i> for more information on the VkMsgApp class and the IRIS ViewKit support for ToolTalk.</p>	Driver
<i>CalcWindow.h</i>	The header file that describes the window class you created as your top-level window.	Top-level window
<i>CalcWindow.C</i>	The source file that implements the window class you created as your top-level window. This is the source file.	Top-level window
<i>CalculatorUI.h</i>	The header file containing the user interface code that describes your application.	Component
CalculatorUI.C	The source file containing the user interface code that implements your application.	Component
Calculator.h	The header file containing the code that describes the derived class from <i>CalculatorUI.h</i> .	Component
Calculator.C	The source file containing the code that implements the derived class.	Component
Calculator	The application resource file	Support
Makefile	The file that describes the dependencies for compilation	Support

Table 5-1 (continued) Files Created by RapidApp

File/Directory Name	Description	Category
calculator.idb	The installation database (idb) file used by Software Packager.	Support
calculator.spec	The specification file used by Software Packager.	Support
icon.fti	The file containing the desktop icon for the application.	Support
desktop.ftr	The file typing rules for the desktop icon used to represent the application.	Support
Calculator.icon	The file containing the icon used when the application is iconified.	Support
unimplemented.C	The file that contains the VkUnimplemented function described in “Using the Debugger to Add Code.”	Support

In most cases, you should only need to edit the various class files to add functionality, the *desktop.ftr* and *icon.fti* file to customize the desktop icon, and the *<Application>.icon* file to customize the minimized window icon.

Class Architecture

To minimize conflicts when it generates code for a component, RapidApp generates two separate C++ classes by default. One class, which normally 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 only need to 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, for greatest separation between the generated code and the code you add by hand, add and change code in only the derived classes (that is, those classes without the "UI" suffix).

You can choose to not use this separation when you create a class, in which case both RapidApp's generated code and any changes you make will be in the same file. The advantage of this approach is that your project will have fewer files. The disadvantage is that there is less separation between the generated code and hand-written code.

Code Merging

When you make changes using RapidApp and then generate new code, RapidApp merges the changes you've made with the code it generates. This allows you to extend the code using your own editor, without losing the ability to evolve the code using RapidApp.

By default, RapidApp uses two independent merge algorithms applied in sequence, the block merge and the three-way merge.

Block Merge

In this first merge RapidApp replaces all sections in the newly generated code that lie between commented editable code blocks with the code from the same section of the current file. The editable code blocks are set off by the comments:

```
//--- Begin editable code block  
  
//--- End editable code block
```

When block merging is used, you can add any amount of code between the comment markers. The code is retained, exactly as-is, each time you regenerate code from RapidApp. RapidApp places editable code blocks in nearly every member function, as well as other strategic places in files. You should be able to limit your changes to these areas, thus ensuring a smooth merge process.

To customize RapidApp's block merge, do the following:

1. From the File menu, choose "Preferences."
2. Go to the Merge Options card.
3. Set or unset the "Do block merge" option.

Note: Although this option is configurable, it would be very unusual to deactivate this portion of the merge algorithm.

Three-Way Merge

In this second merge RapidApp uses a three-way, line-by-line differencing algorithm to detect changes made since the last time code was generated and attempts to silently merge any differences. It is meant to catch any changes you may have made that lie outside of the editable code block comments.

RapidApp applies this merge to four different file types.

UI classes	The base UI classes
Derived classes	The classes derived from the UI classes. If you've chosen not to split classes into UI and Derived classes, this option applies to the single class.
Makefile	The <i>Makefile</i> used to compile your application
Aux files	Auxiliary files such as the application resources file, the desktop icon, the FTR file, and the files used by Software Packager to generate an installable image

To customize RapidApp's three-way merge, do the following:

1. From the File menu, choose "Preferences."
2. Go to the Merge Options card.
3. Specify how RapidApp applies the three-way merge to the four different file types.

Merge RapidApp attempts to merge changes you've made to the existing file with the new file it is generating. The merge process is described below. Typically, this is the best choice if you have made edits outside the editable code blocks, and it is the default for all files.

Don't Merge	RapidApp writes all the new files with a <i>.RapidApp</i> extension. You must then merge the files by hand. You might want to specify this method if you make complex changes to files (for example, creating a highly customized <i>Makefile</i>) and want to be certain that your changes are preserved.
Overwrite	<p>RapidApp overwrites the existing file with the newly generated file. In this case, RapidApp always backs up the old file to the project's <i>.backup</i> directory even if you've unset off the "Save backup" option in the Merge Style card. This is the fastest method of the three.</p> <p>You might use the Overwrite method for Makefiles, UI and Derived classes if you know you are limiting your changes to the editable code blocks. You might also use this method for auxiliary files early in development. When you begin customizing your application's resource file, desktop icon, and files used by Software Packager, you'll want to change to one of the other methods.</p>

When you select the merge method, RapidApp performs the following steps for each file each time you generate code (these steps use *Makefile* as an example):

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.

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 project 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 the 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 between the newly generated file and the current *Makefile* and the previous version, RapidApp extracts all sections that lie between the editable code blocks in the current *Makefile*, and inserts them into the newly generated file. This step is only performed if the “Do block merge” option is set.
6. Next, if there are differences in all three files, and the merge option for this file is “Merge”, RapidApp initiates a three-way merge (see “Three-Way Merge” on page 59, or the *merge(1)* reference page). This merge 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 are resolved successfully, the merged changes are made to *Makefile*, and *.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.
7. 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 make the ancestor files used in the merge process transparent. Each time RapidApp generates code, it also saves a file named *RapidAppData.rap*. This file contains information about the state of the project at the time of this code generation and can be used to provide the ancestor files in the *.buildersource* directory, as well as to provide other information about the project. If you use a configuration management system and are working with multiple users, this file should be placed under source control. You do not need to place the contents of the *.buildersource* directory under source control.

Advanced Topics

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

- “Work With Windows,” gives you tips on how to use top-level windows.
- “Work With Containers,” discusses the advantages and limitations of the different containers available.
- “Work With Menus,” gives instructions for creating menu bars and options menus.
- “Work With Dialogs” describes how to incorporate dialogs in your application.
- “Work With User-Defined 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 A, “RapidApp Reference.”

Work With 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 6-1, RapidApp provides a choice of two primary windows—a Simple Window and a VkWindow—and several types of dialogs. This section describes the features of the primary windows and when it's appropriate to use each type. "Work With Dialogs" on page 97 describes how to incorporate dialogs in your application.

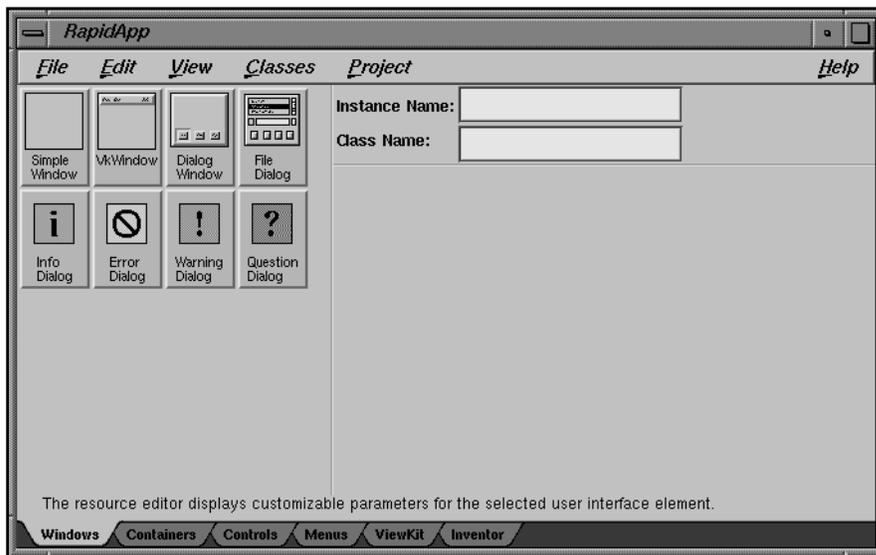


Figure 6-1 The RapidApp Windows Palette

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. However, this shell will not be part of your program. The shell is only a placeholder, necessary to make the container visible in the desktop environment. User interface elements created outside one of the Window containers will be treated as classes, code will be generated, but the object will not be instantiated in the program.

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 69 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.)

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 69 for more information on main and co-primary windows.

Figure 6-2 shows the default configuration of the VkWindow component.

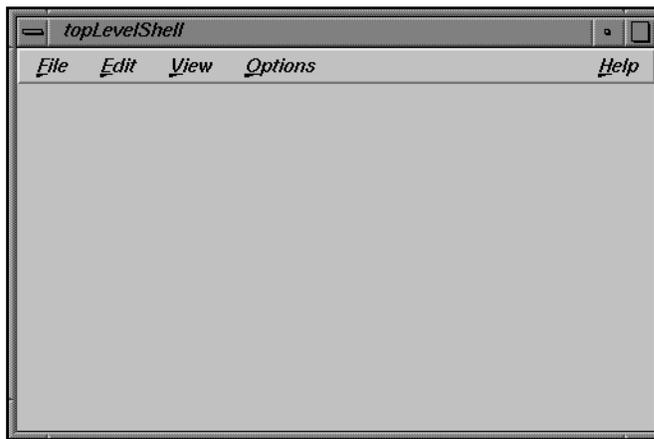


Figure 6-2 Default Configuration of VkWindow Component

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.)

By default, 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. You can disable this feature by setting the window’s **autoRouteCallbacks** resource to “False.”

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 6-1 summarizes the actions and the functions added for the default items on the File and Edit menus.

Table 6-1 Default Actions of Standard `VkWindow` Menu Items

Menu	Selection	Function Added to Child Component or Class	Default Action
File	New	<code>newFile()</code>	The child creates a new, empty file.
	Open	<code>openFile(const char *)</code>	The <code>VkWindow</code> automatically displays a file selection dialog and, if the user selects a file, passes that filename to the child as an argument to the <code>openFile()</code> function. The child opens the given file.
	Save	<code>save()</code>	The child saves its current state to the current file.
	Save As	<code>saveas(const char *)</code>	The <code>VkWindow</code> 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 <code>saveas()</code> function. The child saves its current state to the specified file.
	Print	<code>print(const char *)</code>	The child prints its contents. Currently, the argument to <code>print()</code> is unused.
	Close		The <code>VkWindow</code> deletes itself. To change this behavior, edit the <code>close()</code> function of the <code>VkWindow</code> subclass.
	Exit		The <code>VkWindow</code> calls <code>VkApp:quitYourself()</code> to exit the application. (See Chapter 3, “The ViewKit Application Class,” in the <i>IRIS ViewKit Programmer’s Guide</i> for more information on the <code>VkApp</code> class.) To change this behavior, edit the <code>quit()</code> function of the <code>VkWindow</code> subclass.

Table 6-1 (continued) Default Actions of Standard VkWindow Menu Items

Menu	Selection	Function Added to Child Component or Class	Default Action
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 <i>IRIS ViewKit Programmer’s Guide</i> 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 <i>Indigo Magic User Interface Guidelines</i> for guidelines and Chapter 7, “Interapplication Data Exchange,” in the <i>Indigo Magic Desktop Integration Guide</i> for instructions on implementing cut and paste in your application.
	Copy	copy()	The child copies its current selection to the clipboard. See Chapter 5, “Data Exchange on the Indigo Magic Desktop,” in the <i>Indigo Magic User Interface Guidelines</i> for guidelines and Chapter 7, “Interapplication Data Exchange,” in the <i>Indigo Magic Desktop Integration Guide</i> 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 <i>Indigo Magic User Interface Guidelines</i> for guidelines and Chapter 7, “Interapplication Data Exchange,” in the <i>Indigo Magic Desktop Integration Guide</i> 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 “Work With Menus” on page 92.

Note: Don't remove or edit the Help menu using RapidApp. 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*.

If your application doesn't include online help, you can disable the Help menu by setting the window's `hideHelpMenu` resource to `True`.

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 recommended 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 **copriamaryWindow** resource, as shown in Figure 6-3.



Figure 6-3 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 **copriamaryWindow** 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.

Adding New Top-Level Window Layouts to RapidApp

The `VkWindow` user interface element provided on RapidApp's Windows palette is somewhat different from many other elements on RapidApp's palettes. The `VkWindow` (and `VkSimpleWindow`) elements are not individual widgets, and are not classes. They are simply templates, created as a collection of other widgets, that can be altered and extended to meet your needs. RapidApp treats these windows specially, so that they ultimately become a C++ class, derived from `VkWindow` or `VkSimpleWindow`, but at the beginning, these elements are simply a description of a particular combination of widgets.

The `VkWindow` template has been chosen to conform to the SGI user interface guidelines, and provides menu entries that are often useful. However, you may find that you have additional or simply different menu items that you would like to use repeatedly, and would like to avoid having to start with a `VkWindow` and make the same modifications each time you need a window.

RapidApp allows you to edit a `VkWindow` object once, and save the results as a template that can be used any time you want the same interface. To create and save a template, follow these steps:

1. Start RapidApp and click on the `VkWindow` icon.
2. Edit the window however you like.
3. Select "Save As..." from the File menu.
4. Supply the name of a file in which to save your work, as normal.
5. Before clicking on *OK*, change the option menu on the File dialog to read "Window Template". Then click on *OK*.
6. When a dialog appears, type by which you would like this template to be known on the RapidApp palette. Then click on *OK*.
7. Exit and restart RapidApp. Your template should appear on the Windows palette. Classes created from this template will be derived from `VkWindow` (or possibly other classes) the same as the built-in template. However, the contents of the window are defined by your template, plus any changes you make when using the template.

Work With 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. “Work With User-Defined Components” on page 104 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 6-4.

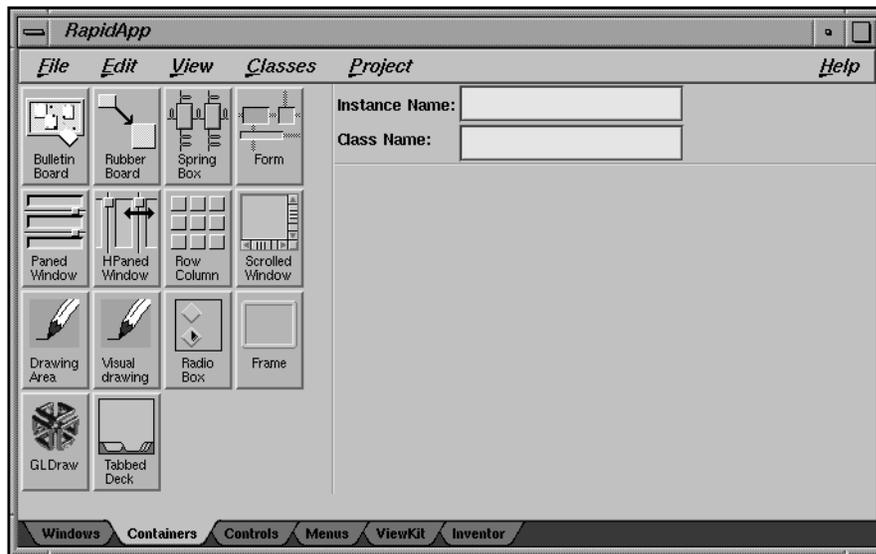


Figure 6-4 RapidApp Containers Palette

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. For best results, make this as small as possible.
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.
7. Toggle on the **autoPosition** resource. 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 6-5 through Figure 6-8 demonstrate the behavior of the Rubber Board widget. To begin, create an interface in a small container, such as in Figure 6-5. Once you've finished the layout, set the Rubber Board's **setInitial** resource to True.

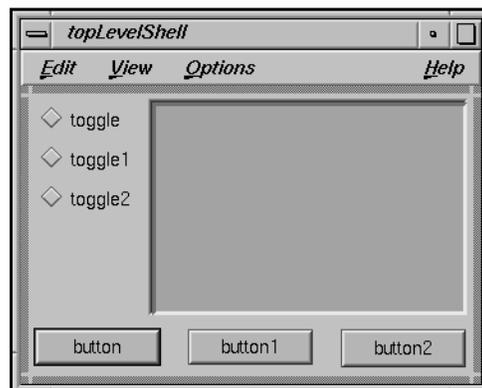


Figure 6-5 Rubber Board: Initial Layout

Next, resize the Rubber Board to a much larger size, as in Figure 6-6. Notice that all widgets keep their original size and position.

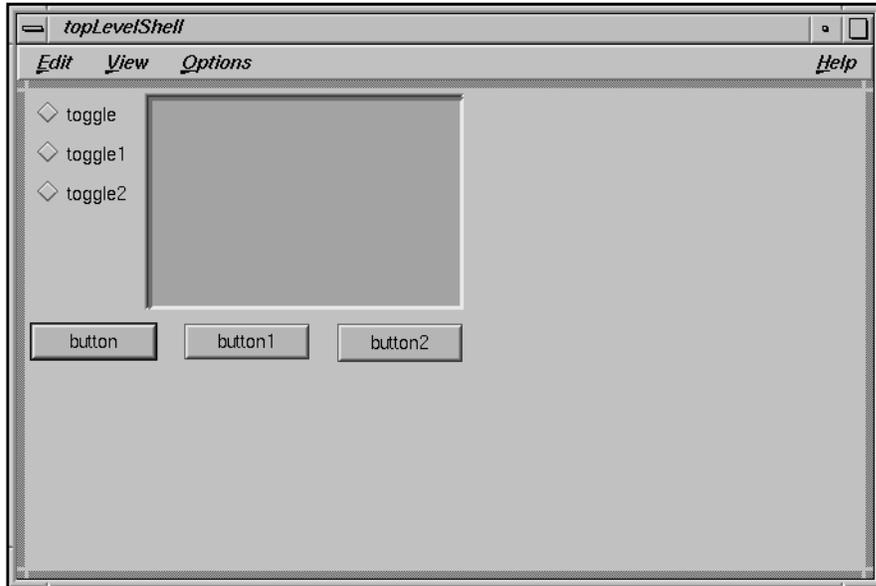


Figure 6-6 Rubber Board: Preparing for Larger Layout

Then resize and reposition the elements to reflect their desired size and position for the larger container size, as shown in Figure 6-7.

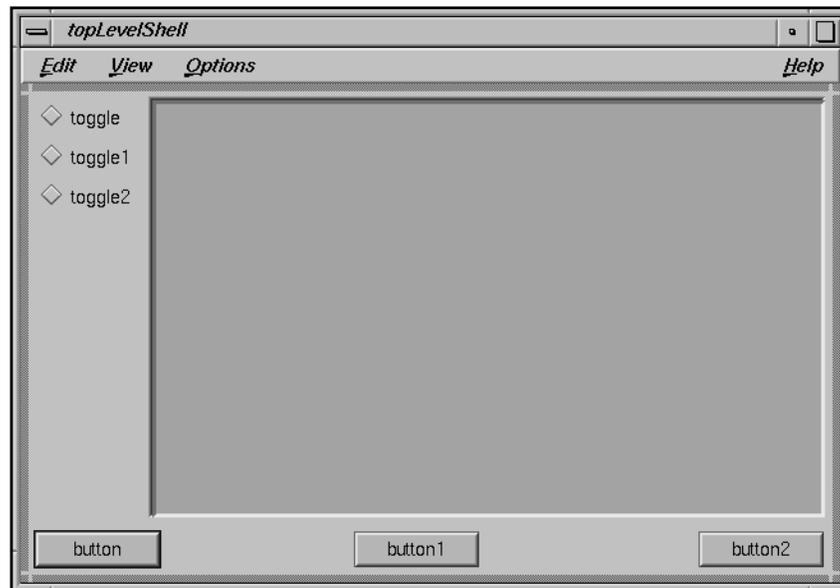


Figure 6-7 Rubber Board: Final Layout

After setting the Rubber Board's **setFinal** and **autoPosition** resources, you can resize the Rubber Board to any shape and the children will maintain their relative positions and sizes, as demonstrated in Figure 6-8.

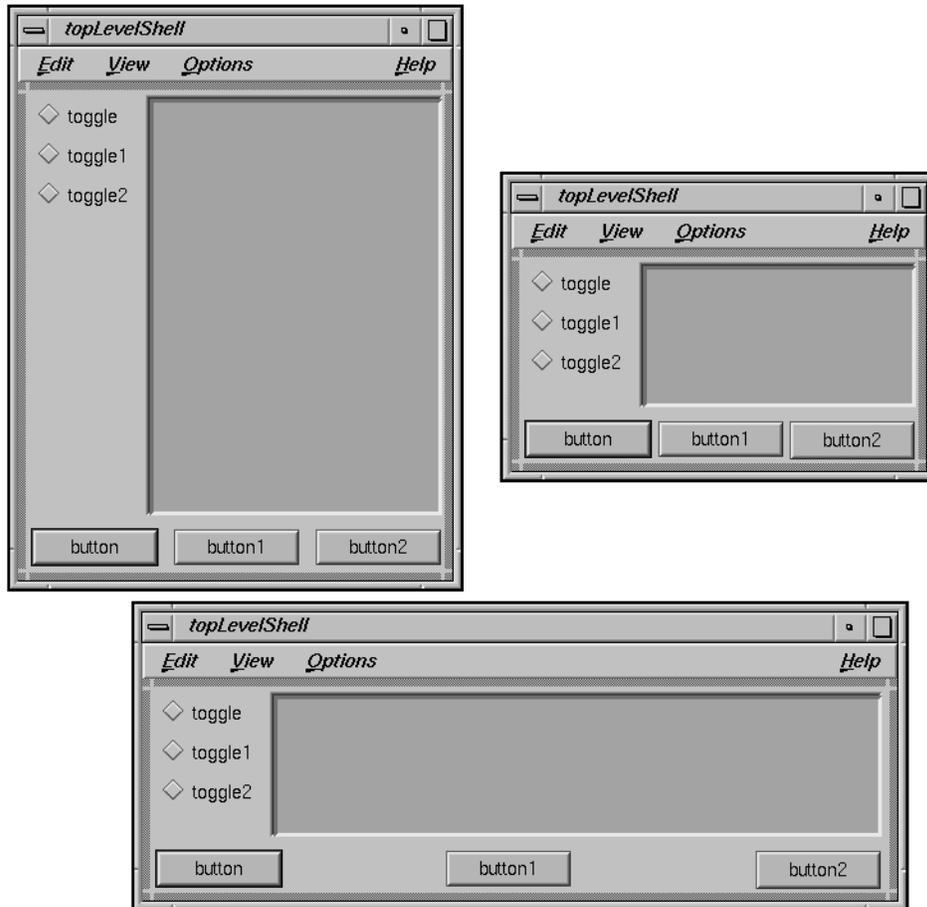


Figure 6-8 Effect of Resizing the Final Rubber Board Layout

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 6-9 shows two simple layouts with buttons placed in vertical and horizontal Spring Boxes.



Figure 6-9 Vertical and Horizontal Spring Boxes

What you can't see from Figure 6-9 is that each child of the Spring Box has six springs associated with it. 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.

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 6-10.

Constraint Resources Added By Parent	
bottomSpring	50
horizontalSpring	100
leftSpring	0
rightSpring	0
topSpring	50
verticalSpring	0

Figure 6-10 Setting Spring Resources

For example, consider the behavior if you set up the spring values as shown in the following table:

Table 6-2 Spring Values

Spring	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 6-11 shows the layout created by these values, both when the Spring Box is its natural size and when it is stretched.

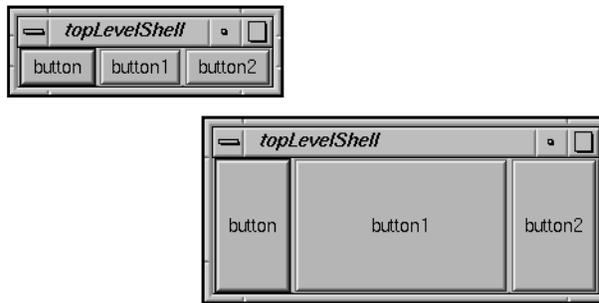


Figure 6-11 Spring Box Behavior With Modified Values

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

As a convenience, the Spring Box container itself includes two resources, **defaultHorizontalLayout** and **defaultVerticalLayout**, to help you create common layouts. Settings these resource causes the Spring Box container to apply a collection of resource settings to all the children it currently contains. Each resource is independent and controls only the resources that apply vertically or horizontally. The meaning of these resources does not change with the value of the **orientation** resource (that is, vertical is always vertical). Possible layouts include:

- XmCENTER** Centers all children in the middle of the Spring Box, with equal spacing on either side of the entire group of children.
- XmSPAN** Stretches all children equally to fill the entire space of the Spring Box.
- XmLEFT** Sets all children to their natural size and moves them to the left edge of the Spring Box. Applies only to the **defaultHorizontalLayout** resource.
- XmRIGHT** Sets all children to their natural size and moves them to the right edge of the Spring Box. Applies only to the **defaultHorizontalLayout** resource.
- XmTOP** Sets all children to their natural size and moves them to the top edge of the Spring Box. Applies only to the **defaultVerticalLayout** resource.
- XmBOTTOM** Sets all children to their natural size and moves them to the bottom edge of the Spring Box. Applies only to the **defaultVerticalLayout** resource.
- XmDISTRIBUTE**
Sets all children to their natural size and distributes them evenly across any open space in the Spring Box.
- 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 child's spring resources.

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 6-12 shows what happens when you add a push button to a Form.

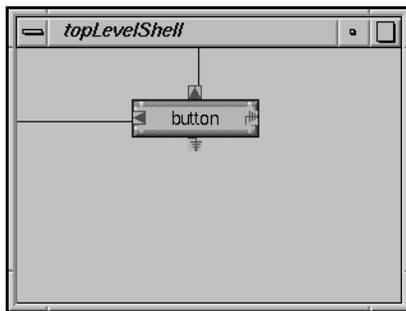


Figure 6-12 Push Button in a Form

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.

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 6-13 sets the top offset to zero.

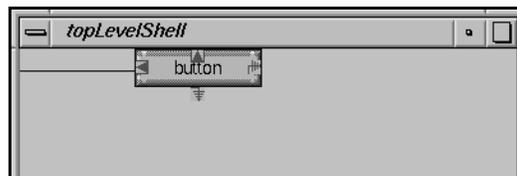


Figure 6-13 Setting the Top Offset to Zero

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 6-14 shows an example.

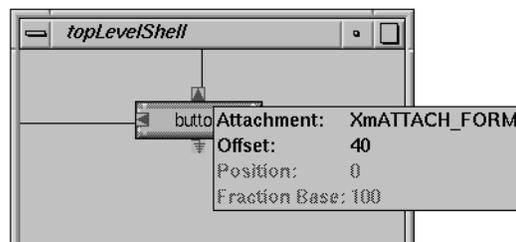


Figure 6-14 Using the Popup to Set an Offset

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 6-15 demonstrates pressing the right mouse button over the right attachment icon. Figure 6-16 shows the results of selecting `XmATTACH_FORM`.

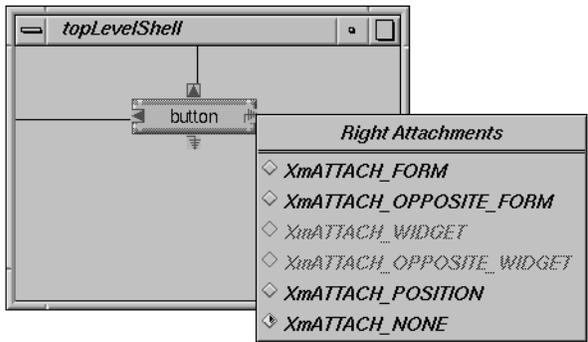


Figure 6-15 Displaying the Attachment Menu

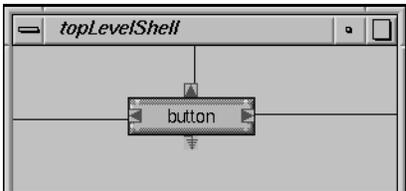


Figure 6-16 Push Button With a Right Attachment

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 6-17.

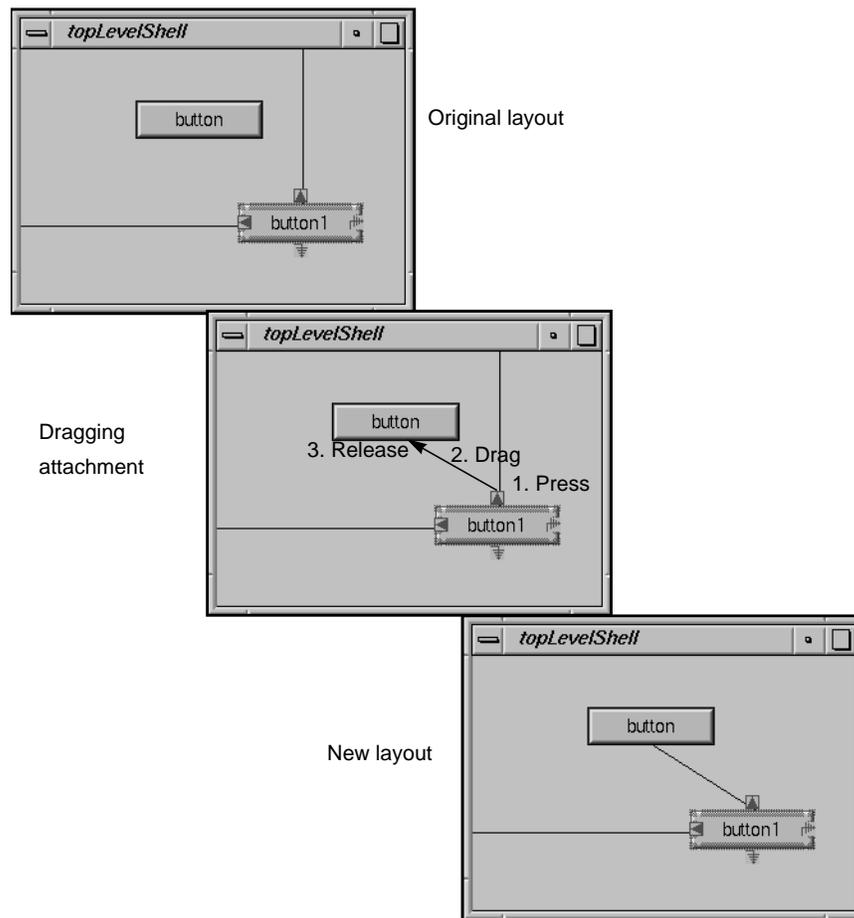


Figure 6-17 Drawing an Attachment

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 6-18 shows an example of the HPaned Window.

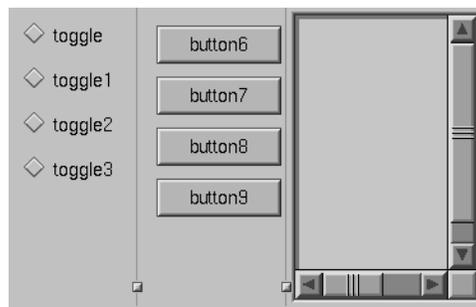


Figure 6-18 HPaned Window Container

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 6-19, then the RowColumn widget is a good choice. Otherwise, you might want to choose another container.



Figure 6-19 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 its 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 6-20.



Figure 6-20 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 88 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 6-21. You can change its position slightly by editing the label’s constraint resources added by the Frame.

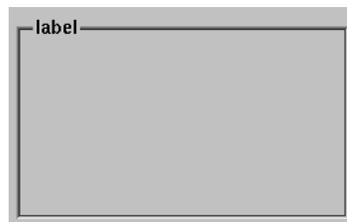


Figure 6-21 Frame Widget

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 three drawing area widgets: Drawing Area (**XmDrawingArea**), Visual Drawing (**SgVisualDrawingArea**), and GLDraw (**GLwMDrawingArea**). The Drawing Area and Visual Drawing 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.

Note: Although both the Drawing Area and Visual Drawing 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. See the `GLwMDrawingArea(3Xm)` reference page for more information on the **GLwMDrawingArea** widget.

Tabbed Deck

The Tabbed Deck component (**VkTabbedDeck**) is a special container that 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 6-22 shows an example of a Tabbed Deck.

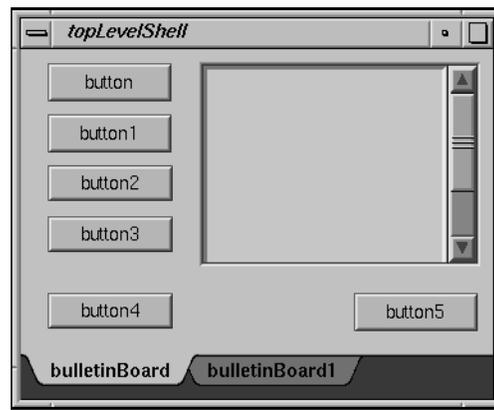


Figure 6-22 Tabbed Deck

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.

Work With Menus

The menus palette, shown in Figure 6-23, allows you to create menus and menu items. RapidApp allows you to create and manipulate both menu bars and option menus.

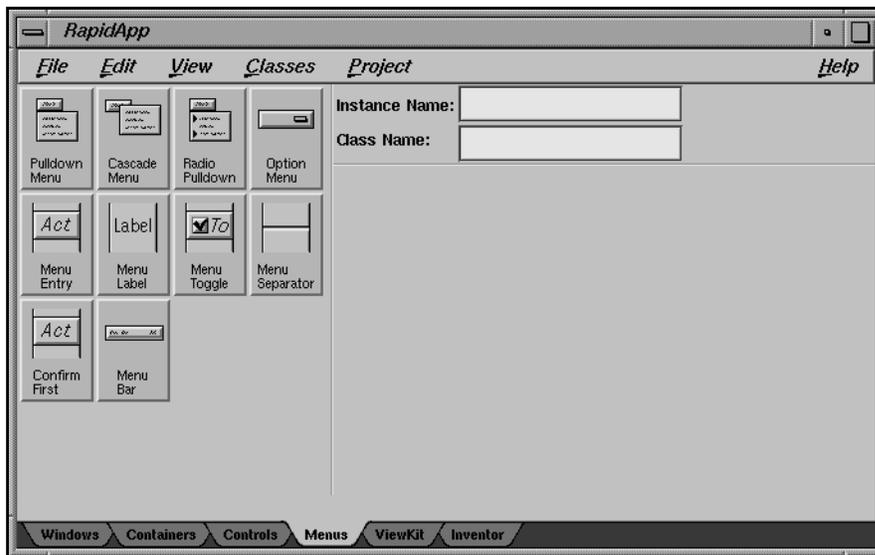


Figure 6-23 RapidApp Menus Palette

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 94 for information on editing the contents of individual menu panes.

Creating a Menu Bar

The only way to create a menu bar on a main window 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 65 for more information on the standard menu bar entries.

You can add a menu bar to a dialog if you want a custom dialog. To do so, select the Menu Bar element from the Menus palette and add it to the desired dialog as you would any other element. This menu bar doesn't contain the standard menu bar entries described in “`VkWindows`” on page 65; instead, it contains only two dummy menu panes with three dummy menu entries each. See “Creating and Using Custom Dialogs” on page 99 for more information on creating custom dialogs.

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 94.

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 94.

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 <i>radio behavior</i> 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 <i>radio behavior</i> (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 94.

Work With Dialogs

By comparison to other parts of your interface, there is little to customize for most dialogs. In fact, IRIS IM includes standard dialogs for uses such as warnings, errors, and file selection. 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.

As a result, standard dialogs are not well suited for construction with RapidApp. In most cases, you should use the dialog management system provided by IRIS ViewKit. (For more information on the IRIS ViewKit dialog mechanism, see Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide*.) However, RapidApp supports a method for creating customized dialogs that use the IRIS ViewKit dialog management system.

This section describes:

- using the IRIS ViewKit dialog management system for standard dialogs
- creating and using custom dialogs with RapidApp

Using the IRIS ViewKit Dialog System for Standard Dialogs

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 multiwindow 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. IRIS ViewKit encapsulates dialog management, including caching, in the abstract **VkDialogManager** class that serves as a base class for other, specific dialog classes. Each type of dialog in IRIS ViewKit has a separate class derived from **VkDialogManager**. Each class is responsible for managing its own type of dialog (for example, each class maintains its own dialog cache).

The header file for each dialog class provides a global pointer to the instance of that class's dialog manager. The name of the pointer consists of "the" followed by the dialog type. For example, the global pointer to the information dialog manager declared in `<Vk/VkInfoDialog.h>` is *theInfoDialog*, the global pointer to the error dialog manager declared in `<Vk/VkErrorDialog.h>` is *theErrorDialog*, and so forth. To access the dialog managers in your application, simply use these global pointers.

The **VkDialogManager** class offers four different functions for posting dialogs:

- post()** Posts a non-blocking, non-modal dialog. The function immediately returns, and the application continues to process user input in all windows.
- postModal()** Posts a non-blocking, full-application-modal dialog. The function immediately returns, but the user cannot interact with any application windows until after dismissing the dialog.
- postBlocked()** Posts a blocking, full-application-modal dialog. The user cannot interact with any application windows until after dismissing the dialog. Furthermore, the function does not return until the user dismisses the dialog.
- postAndWait()** Posts a blocking, full-application-modal dialog. The user cannot interact with any application windows until after dismissing the dialog. Furthermore, the function does not return until the user dismisses the dialog. **postAndWait()** is simpler to use than **postBlocked()**, but it does not allow as much programming flexibility.

Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. For more information on the IRIS ViewKit dialog mechanism, see Chapter 7, "Using Dialogs in ViewKit," in the *IRIS ViewKit Programmer's Guide*. You can also consult the `VkDialogManager(3Vk)` reference page for more information on the functions provided by the **VkDialogManager** class.

Creating and Using Custom Dialogs

Occasionally, you might need to create a custom dialog not implemented in IRIS ViewKit. You can use any of the dialogs on the Windows palette as the basis for a custom dialog. Any custom dialog that you create with RapidApp integrates with the IRIS ViewKit dialog mechanism; you post, dismiss, and otherwise interact with them in your application in the same way as the predefined IRIS ViewKit dialogs.

Selecting a Dialog to Customize

You can use any of the standard dialogs on the Windows palette as the basis for a custom dialog. If none of the predefined dialog types are appropriate, you can start with the Dialog Window element. Figure 6-24 shows the default configuration of a Dialog Window. The Dialog Window is ideal for implementing *support windows*, which are described in Chapter 3, “Windows in the Indigo Magic Environment,” in the *Indigo Magic User Interface Guidelines*.

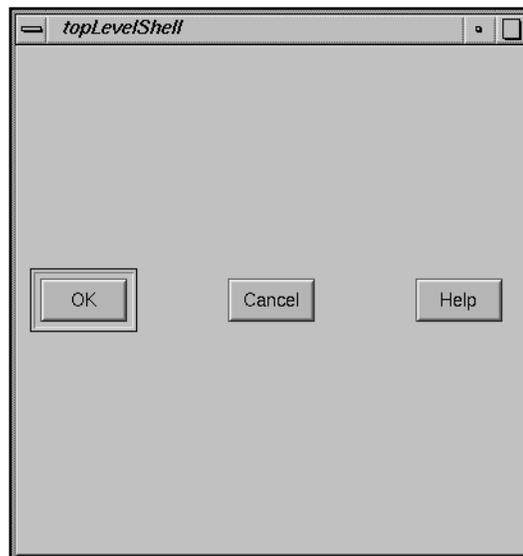


Figure 6-24 Default Configuration of Dialog Window

Adding Elements to the Dialog

In addition to the standard elements provided by the dialog you select to begin with, you can add one child element. The child element that you place in a dialog is typically either a container widget or a complex component.

Adding a Menu Bar to the Dialog

You can also add an optional menu bar to your customized dialog. To do so, go to the Menus palette, select the Menu Bar element, and add it as a child of the dialog. You can then edit the Menu Bar as described in “Work With Menus” on page 92.

Setting the Dialog Message, Dialog Title, and Button Labels

In the IRIS ViewKit dialog mechanism, you set the dialog message, dialog title, and button labels using **VkDialogManager** member functions before posting each dialog. Because you could specify different messages, titles, and button labels when you post the dialog from different points in your program, RapidApp doesn’t offer any method of setting these items. Instead, you must use the appropriate **VkDialogManager** member functions when you post the dialogs.

The **post()**, **postModal()**, **postBlocked()**, and **postAndWait()** functions each accept as an argument the message for the posted dialog. The **setTitle()** function sets the title for the posted dialog. The **setButtonLabels()** function sets the button labels for the posted dialog. See the **VkDialogManager(3Vk)** reference page and Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide* for more information on setting the dialog message, dialog title, and button labels.

Code Generation for Customized Standard Dialogs

When RapidApp generates code for a customized standard dialog, it creates the dialog as a subclass of the corresponding IRIS ViewKit dialog class. For example, if you customize a File Selection dialog, RapidApp creates a subclass of **VkFileSelectionDialog**. (See Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide* for more information on the IRIS ViewKit dialog classes.)

For customized standard dialogs, RapidApp creates all child elements (including the menu bar, if you’ve added one to the dialog) directly in the dialog subclass’s **createDialog()** member function. RapidApp does this, rather than encapsulating the child elements in a subclass of **VkComponent** as it does for other windows, because

when you customize a standard dialog, it's usually to add an option menu, a few toggle buttons, or some other minor addition. Encapsulating these elements in a component—and generating four extra source and header files in the process—is overkill in this case. If you add a more complex set of controls to a standard dialog, you can always explicitly encapsulate these controls in a class as described in “Creating Components” on page 105.

For customized standard dialogs, RapidApp includes in the dialog subclass the callbacks associated with all menu items, pushbuttons, and other controls that you add to your customized dialog.

How you specify the actions taken when the user clicks a standard dialog button (that is, the *OK*, *Apply*, and *Cancel* buttons) depends on the function you use to post the dialog. If you post the dialog using `post()`, `postModal()`, or `postBlocked()`, you provide a callback function for each button. If you post the dialog using `postAndWait()`, you test the enumerated value returned by the function and then perform an appropriate action. See “Posting Dialogs” in Chapter 7 of the *IRIS ViewKit Programmer's Guide* for more information on specifying actions to take when users click dialog buttons.

Code Generation for Dialogs Customized From the Dialog Window

When RapidApp generates code for a dialog customized from the Dialog Window, it creates the dialog as a subclass of `VkGenericDialog`. See “Deriving New Dialog Classes Using the Generic Dialog” in Chapter 7 of the *IRIS ViewKit Programmer's Guide* for more information on the `VkGenericDialog` class.

For Dialog Windows, if the child element of the customized dialog 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 customized dialog then simply creates an instance of this class in its `createDialog()` member function.

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

RapidApp creates the `VkComponent` subclass in this case (rather than simply creating the child elements directly in the `createDialog()` member function as it does for customized standard dialogs) because Dialog Windows usually contain palettes or other complex controls typical of *support windows*. In these cases, it makes sense to follow the same encapsulation scheme used by the Simple Window and `VkWindow` components.

For customized Dialog Windows, RapidApp includes the callbacks associated with all menu items, pushbuttons, and other controls that you add to your customized dialog in the **VkComponent** subclass it generates to contain the child elements of the dialog.

How you specify the actions taken when the user clicks a standard dialog button (that is, the *OK*, *Apply*, and *Cancel* buttons) depends on the function you use to post the dialog. If you post the dialog using **post()**, **postModal()**, or **postBlocked()**, you provide a callback function for each button. If you post the dialog using **postAndWait()**, you test the enumerated value returned by the function and then perform an appropriate action. See “Posting Dialogs” in Chapter 7 of the *IRIS ViewKit Programmer’s Guide* for more information on specifying actions to take when users click dialog buttons.

RapidApp also generates **ok()**, **cancel()**, and **apply()** functions in the dialog subclass. The default actions of these functions are to call their corresponding **VkDialogManager** functions, which dismiss your dialog whenever the user clicks on either the *OK* or *Cancel* button, and keeps the dialog posted whenever the user clicks on the *Apply* button. You can change this behavior by editing these functions in your dialog subclass. You can also include in them any other code needed to support your custom dialog (for example, storing the values of various controls in class data members that you can then retrieve using access functions that you provide for your class).

Example of a Custom Dialog

Figure 6-25 shows an example of a custom dialog created from the Dialog Window. 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 6-25 Example of a Custom Dialog

When RapidApp generates the code for the dialog's child class, it adds three member functions to it: **ok()**, **cancel()**, and **apply()**. 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 6-25, 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 6-25 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 6-25. 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();
```

Work With User-Defined Components

An important concept in RapidApp is creating self-contained components—C++ gui classes—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. See Chapter 8, “Component Libraries,” for information on how to create and use libraries of components that you can reuse in other applications and distribute to other developers.

Creating Components

It's easy to create components in RapidApp.

1. Select the container that you want to be the top-level element in your component
2. From the Classes menu, choose "Make Class."

RapidApp displays a dialog prompting you for the name of your new class, as show in Figure 6-26.

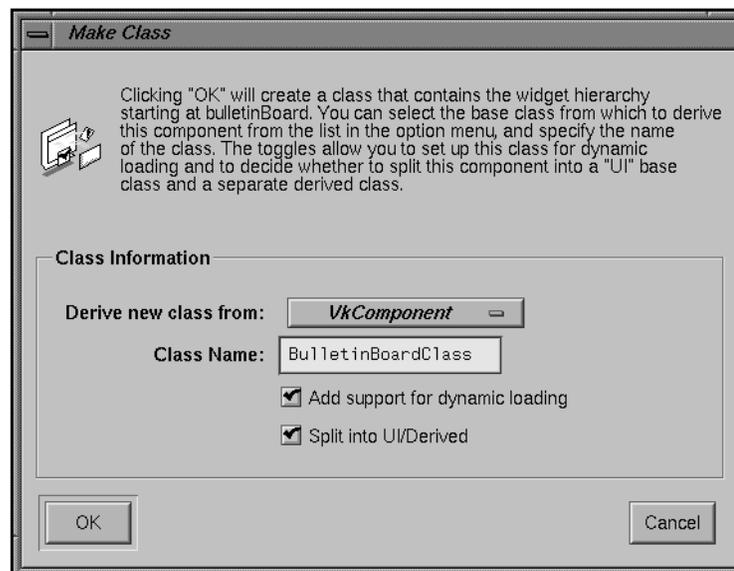


Figure 6-26 Make Class Dialog

3. Determine which class the component should be derived from.

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, choose the IRIS ViewKit **VkMsgComponent** option.

You can also include your own super class in this option menu. For information on how to do this, see "Deriving From Your Own Super Class" on page 109.

Note: To handle ToolTalk messages, you also need to select “Preferences” from the File menu, go to the RapidApp card, and set the Message system to “ToolTalk.” 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.

4. Keep or change the Class Name.
5. Set or unset the “Add support for dynamic loading” option.

RapidApp allows you to create components that can be loaded back onto the RapidApp palette as first class objects. To support this, RapidApp adds two simple functions to each class, to allow them to be dynamically loaded.

Note: This option is set by default for non-window components and meaningless for SimpleWindow, VkWindow, and dialog classes. To globally set this option, go to the Code Style card in the Preferences dialog and set the “Add support for dynamic loading” option.

6. Set or unset the “Split into UI/Derived” option.

As discussed in “Class Architecture” on page 57, by default, 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. If you use this approach, when you add the functional code to your component, you only need to do so in the derived class.

Note: This option is set by default for non-window components and isn’t meaningful for SimpleWindow, VkWindow, and dialog classes. To globally set this option, go to the Code Style card in the Preferences dialog and set the “Split classes into UI/Derived” option.

7. Click OK.

After creating the new class, RapidApp adds its representing icon to the User-Defined palette. (RapidApp creates the palette if it doesn’t already exist.)

An Example: Creating a Component

As an example of creating a component, consider the calculator program created in Chapter 1, “RapidApp Tour.”. 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 6-27.

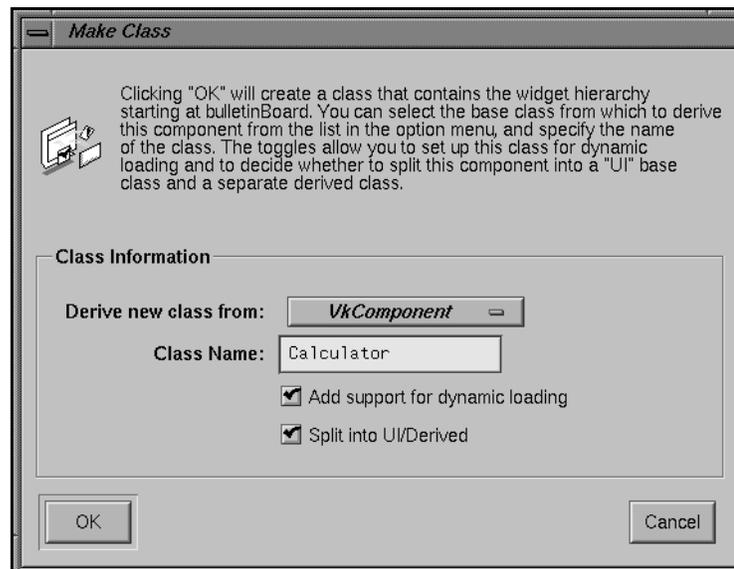


Figure 6-27 Creating a Calculator Class

RapidApp updates the resource editor area and header area to display information about the **Calculator** class that you created. The header information is shown in Figure 6-28.

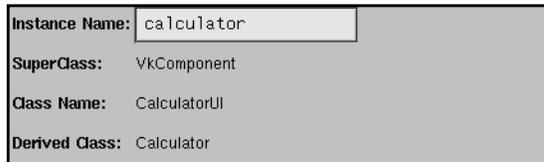


Figure 6-28 Class Header

RapidApp also creates a new palette named “User-Defined” (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 6-29. You can now create additional instances of the **Calculator** component just as you would any other interface element.



Figure 6-29 Calculator Icon from User-Defined 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 a 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.

Deriving From Your Own Super Class

You can provide your own classes to be used as super classes for simple components, normally subclasses of `VkComponent`, and windows, which can be subclasses of `VkWindow` or `VkSimpleWindow`. You cannot currently define super classes for Dialogs. There are several restrictions:

- Your class must be derived directly, or indirectly from `VkComponent`, `VkWindow`, or `VkSimpleWindow`.
- Your new base class must not have any UI itself. The mechanism supports extensions to the API, not visual differences.
- The class should not be abstract, or at least be aware that if a class is abstract, classes derived from it will not be usable until you implement the pure virtual member functions.

The set of base classes available for windows without menus (derivations of `VkSimpleWindow`) is defined by files installed in `/usr/lib/RapidApp/BaseClasses/VkSimpleWindow`

The set of base classes available for windows with menus (derivations of `VkWindow`) is defined by files installed in `/usr/lib/RapidApp/BaseClasses/VkWindow`

The set of base classes available for regular components (derivations of `VkComponent`) is defined by files installed in `/usr/lib/RapidApp/BaseClasses/VkComponent`

There are also personal directories under `$HOME/.rapidappdir/BaseClasses`. Each of these directories contain files that provide various information about base classes. For example, the `VkComponent` directory contains a file `VkComponent`, which is the default. This file contains the following:

```
*VkComponent.headerFile: <Vk/VkComponent.h>
*VkComponent.library: -lvk
```

To add a new candidate super class that can be used instead of `VkComponent`, replicate this file format for your class. For example, assume you have a new class, `MyComponent`, defined as:

```
//////////  
// MyComponent.h  
class MyComponent : public VkComponent {  
  
    public:  
        MyComponent(const char *name);  
  
        void print();  
};
```

Compile your class, put it in a library, such as `libmyComponent.so`, and put the header somewhere, perhaps `/usr/include/MyStuff/MyComponent.h`.

Now, create a file named `MyComponent` that contains

```
*MyComponent.headerFile: <MyStuff/MyComponent.h>  
*MyComponent.library: -lmyComponent
```

and put this file in either `/usr/lib/RapidApp/BaseClasses/VkComponent/MyComponent`

or put it under `$HOME/.rapidappdir/BaseClasses/VkComponent`.

Restart `RapidApp`, and the next time you create a class, you should see this class added to the list of base class choices in the popup dialog after you invoke the “Make Class” command.

A similar procedure can be used to define new classes for `VkSimpleWindow` or `VkWindow`.

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 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 redisplay your current interface, reflecting the changes you made to your components.

Deleting Components

Once you create a class, it remains on the User-Defined 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 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.

VkEZ Library

This chapter provides details about the VkEZ library and its EZ convenience functions. It includes the following sections:

- “EZ Convenience Functions”
- “VkEZ Operators”

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 Chapter 1, “RapidApp Tour,” 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 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 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) = (int) EZ(_value1) + (int) 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) = ""; // Clear the text field
EZ(_result) << "The result of " << EZ(_value1) << " + "
            << EZ(_value2) << " = "
            << (int) EZ(_value1) + (int) 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 “VkEZ Operators.”

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 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 in the function invoked when `_dial1` changes value:

```
EZ(_dial2) = (int) 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";
```

For more information on the VkeZ features, see “VkeZ Operators.”

VkeZ Operators

VkeZ has two kinds of operators:

- General operators, which 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";
```

The description of each operator lists the widgets that support that operator and defines how the operator works on each widget.

General Operators

String()

String() 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

int()

int() 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 *);
```

Assignment 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, 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 old contents

Append Operators

```

EZ& operator+=(int);
EZ& operator+=(float);
EZ& operator+=(const char *);
EZ& operator<<(int);
EZ& operator<<(float);
EZ& operator<<(const char *);

```

Append 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 operators for each widget that supports them.

XmToggle, XmToggleButtonGadget	increments the current value of XmNset by the specified value, so <pre>EZ(toggle) += 0;</pre> does nothing, while <pre>EZ(toggle) +=1;</pre> sets <i>toggle</i> 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);
```

The decrement operator decrements the current value associated with a widget.

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

The following are attributes that can be modified using VkeZ:

- Attributes supported by all widgets: border, width, height, x, y
- Attributes supported by all widgets that have setting support for Pixel or char *: background, foreground
- Attributes supported by XmLabel, XmLabelGadget and subclasses: label
- Attributes supported by XmScale, XmScrollBar, SgDial: value, minimum, maximum

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;
```

Component Libraries

So far, this manual has described how to use RapidApp to create complete applications. However, you can also use RapidApp to create libraries of components that you can reuse and share with other developers. This chapter describes how to create component libraries, how to load components onto RapidApp palettes from component libraries, and how to import components into RapidApp that you didn't create in RapidApp.

You can also arrange for RapidApp to create components that are subclassed from classes you have written, in addition to the `VkComponent`, `VkSimpleWindow`, and `VkWindow` classes provided by ViewKit. This chapter describes how to integrate such classes with RapidApp.

Work With Component Libraries

Introduction

By default, RapidApp is configured to create a standalone application, and all of the techniques described so far in this manual (for example, starting by creating top-level windows) assume that you are creating an application. As a result, the *Makefile* generated by RapidApp creates a single executable file; any components that you create for your application are compiled and included in the application rather than in a separate library. Furthermore, the auxiliary files, for example those generated for use with Software Packager, assume a target application rather than a library.

This section describes how to configure RapidApp to create a component library and how to generate the component library. It also provides a simple example.

Configuring RapidApp

Generating a component library with RapidApp rather than an application requires you to follow just a few simple steps:

1. Configure RapidApp to generate a library instead of applications. To do so:
 - From the File menu, choose “Preferences” to display the Preferences dialog.
 - In the Project card, provide the name of a library in the Library name text field. Use the standard library naming format, such as “libXYZ,” but don’t provide a suffix. RapidApp ignores any suffix you include.
 - Provide the name of a header directory in the Library headers text field. Enter the name of a subdirectory relative to */usr/include* (but don’t include “/usr/include” as part of the directory name). When you install your library on your system or create an installable version of your library for other developers, your library’s header files are installed in this subdirectory of */usr/include*. For example, IRIS ViewKit places its headers in */usr/include/Vk*, whereas IRIS IM places its header files in */usr/include/Xm*.
2. Create your components. Normally, you can create them directly on the desktop. You don’t need to create them as children of Simple Window or VkWindow objects, although you can if you wish.
3. Place default resources in classes rather than an application resource file. This encapsulates the default resource with the components rather than making them dependent on an external resource file. To do so:
 - From the File menu, choose “Preferences” to display the Preferences dialog.
 - In the Code Style card set the “Place resources in classes” option.
4. (Optional) If you create components as children of Simple Window or VkWindow objects, you probably want to prevent RapidApp from automatically adding callback functions to your component to support the pass-through of menu callbacks (described in “VkWindow” on page 180). Normally, this feature isn’t useful for a standalone component. To do so:
 - Select the window object containing your component.
 - Set the window’s **autoRouteCallbacks** resource to False.

Build a Component Library

Once you have configured RapidApp as described in “Configuring RapidApp” and created some components, you can generate code and build a component library. This section includes the following:

- “Generating Code for the Component Library”
- “Building the Component Library”
- “Example of Creating a Component Library”

Generating Code for the Component Library

To generate code for the library select “Generate C++” from the Project menu, just as if you were generating code for an application.

RapidApp generates all the files it would for an application. However, the *Makefile* that RapidApp generates differs in several ways from the *Makefile* it would generate for an application:

- It contains as targets both the static and shared libraries.
- The application target links with the library rather than compiling the components directly into the application.
- It contains an “install” target that installs the libraries in */usr/lib* and the class headers in the subdirectory of */usr/include* specified by the Library headers text field of the Project card in the Preferences dialog, but doesn’t install the application itself or any of its support files (for example, its resource file).

Furthermore, the files used by Software Packager for creating an installable image differ in several ways from those generated for an application:

- They install the libraries in */usr/lib*.
- They install the class headers in the subdirectory of */usr/include* specified by the Library headers text field of the Project card in the Preferences dialog.
- They don’t include the application, its resource file, its icon, or its FTR file for installation.

Building the Component Library

When you select “Build Application” from the Project menu, RapidApp creates both a static library (a library that is statically bound to a program compiled with it) with a *.a* suffix and a shared library (a library that binds with a program at run time) with a *.so* suffix. The library includes both the UI classes and their derived subclasses.

Note: RapidApp doesn’t include dialogs, top-level windows, or application objects (instances or subclasses of **VkApp**) in the library.

RapidApp also builds an application, but you should think of this application as a test driver rather than a full application. RapidApp links the component library with the application rather than compiling the components directly into the application. This application displays windows only if you didn’t unset the “Create windows for orphaned objects” option in the RapidApp card of the Preferences dialog (as described in “Configuring RapidApp”).

If you try to run the application from the command line rather than by selecting “Run Application” from the Project menu, you may get an error message such as:

```
7054:calculator: rld: Fatal Error: cannot map soname 'libcalc.so' using any of
the filenames /usr/lib/libcalc.so:/lib/libcalc.so
```

This is because the application needs the shared library (*libcalc.so* in this example) to run. This library is loaded at run time by *rld*, the run-time loader. By default, *rld* looks in */usr/lib* but not in the current directory. If you want to run the program from a shell, there are two approaches you can take:

- Install the library by entering **make install** in a shell. This places the shared library in */usr/lib*. (You might need root permission to do this). This step also installs header files, which you might or might not want to do. Entering **make install** doesn’t install the application.
- Set the environment variable `LD_LIBRARY_PATH` to `“.”` to change the search path *rld* uses to include the current directory. This allows you to run the application without installing libraries or headers on your system.

Notice that if you use the first option, you need to reinstall the library any time you change it for the changes to be reflected in the test application. If you run the application from RapidApp, or set the `LD_LIBRARY_PATH` before running the program from a shell, changes are reflected as soon as you rebuild the library.

Example of Creating a Component Library

As an example of creating a component library, consider the **Calculator** component as created in “Creating Components” on page 105. After creating the **Calculator** component, you can create a library containing it by following these steps:

1. From the File menu, choose “Preferences” to display the Preferences dialog.
2. In the Project card, enter a library name in the Library name text field. In this case, enter “libcalc.”
3. Enter the name of a header directory in the Library headers text field. In this case, enter “CalcLib” so that the header files for the **Calculator** component appear in */usr/include/CalcLib*.
4. Select “Generate C++” from the Project menu to generate code for the component.
5. Select “Build Application” from the Project menu to compile the static and shared libraries. The libraries contain both the **Calculator** and **CalculatorUI** classes.

Install a Component Library

You can install the component library and associated headers on your system simply by entering `make install` in a shell window. You can then use your library as you would any other library.

Note: You don’t need to install the library and headers on your system to create an installable image for distribution to others.

Package a Component Library for Distribution

Once you have created and tested your library, you can use Software Packager to package your library and associated class headers for installation by others. To do so, select “Edit Installation” from the Project menu to launch Software Packager, just as if you were packaging an application for distribution.

RapidApp automatically generates files for Software Packager so that it installs the libraries in */usr/lib* and the class headers in the subdirectory of */usr/include* specified by the Library headers text field on the Project card in the Preferences dialog. Neither the test application nor any of its associated support files (for example, its resource file, its icon, or its FTR file) are included in the installable image. You can either use Software

Packager to generate an installable image for distribution, or you can simply enter `make images` in a shell window in your project directory. For complete instructions for using Software Packager, consult the *Software Packager User's Guide*; Chapter 1, "Packaging Software for Installation: An Overview."

Load Components Onto Palettes

This section includes the following:

- "Introduction"
- "Loading Components Into RapidApp"
- "Deleting a Component from a Custom RapidApp Palette"
- "Creating an Installable Image"
- "Adding Resources to Components"
- "Arguments and Argument Types"
- "Support for Callbacks"

Introduction

Once you create component libraries and install them on your system, you can load them onto RapidApp palettes and use them as you would any other component. These components that you load into RapidApp are "live": they actually operate when RapidApp is in play mode. You can even extend the components to add resources that you can set and change using the RapidApp resource editor.

Using this feature of RapidApp, you can create entire custom palettes of user-created components that you can use to create applications. For example, if your organization has a group producing common services for use by others in creating applications, this group could develop various user interface elements to be used throughout one or more projects and make the elements available to the other groups as libraries of components.

You can also use the techniques described in this section to load some components that you didn't create with RapidApp. However, externally created components must follow several guidelines for RapidApp to be able to load them. "Load Non-RapidApp Components" on page 143 describes how to write an external component so that you can load it onto a RapidApp palette.

Note that loading components differs from simply using “Import” from the File menu to import one or more components in two ways:

- The “Import” option loads only the *.uil* file description of the components, not any functional code that you have added. You can’t access any of the functionality of the components that you add unless you also copy all of the header and source files for those components to your project directory.
- When you generate code, RapidApp regenerates code for the components you import. If you have copied the source and header files for those components to your project directory, RapidApp integrates any code changes with the source files. With “live” components, RapidApp simply links with the library that contains the component.

Loading Components Into RapidApp

This section describes how to load a component onto a RapidApp palette, gives an example of doing so, and describes the files generated when you do so.

Components that you create with RapidApp are normally very easy to load onto a RapidApp palette. You can load any self-contained component, with the exception of top-level windows, such as *VkWindow* or *Simple Window*, and dialogs. Before loading a component into RapidApp, you should consider testing it using the Component Tester application shipped with RapidApp, as discussed in “Testing Components” on page 147.

The following sections are included:

- “Loading a Component Onto a RapidApp Palette”
- “Example of Loading a Component Onto a RapidApp Palette”
- “Files Generated When Loading Components on a Palette”

Loading a Component Onto a RapidApp Palette

Before loading a component onto a RapidApp palette, you must create and install a shared library containing that component, as described in “Work With Component Libraries” on page 123.

Note: You must be sure that all classes you wish to load into RapidApp are marked as “Dynamic” classes. This option is available when you initially create the class as well as in the Class Header when working in Class Edit mode. All classes are dynamic by default, except for subclasses of `VkSimpleWindow`, `VkWindow`, and dialogs.

Once you have created the library, you can add any component it contains by following these steps:

1. Select “Install Classes” from the Classes menu to display the RapidApp Component Importer dialog, as shown in Figure 8-1.

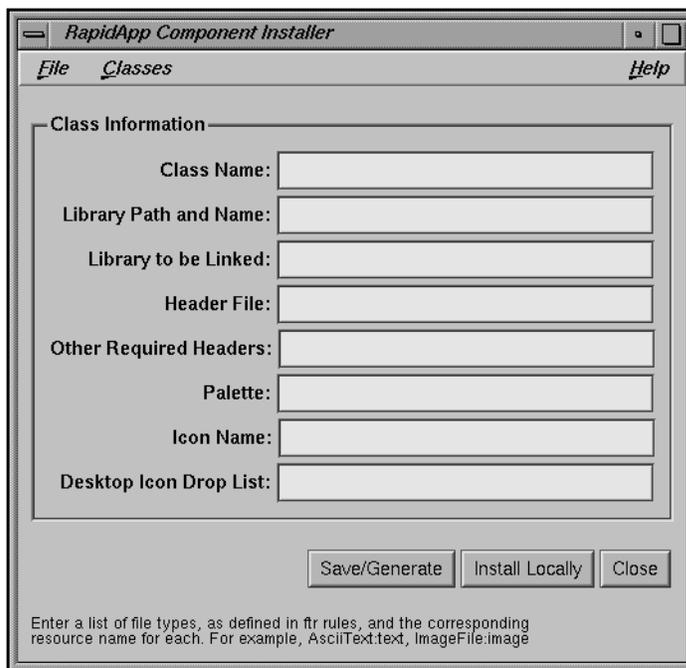


Figure 8-1 The RapidApp Component Importer Dialog

2. In the dialog, fill in each text field. If you are currently working on the project that contains the components in RapidApp, all fields will be filled in with initial values for one of the classes. Other classes can be accessed by selecting the "Browse..." menu item from the "Classes" menu. The values provided are defaults, and can be changed if needed. You need to provide:

Class Name The name of the component to import

Library Path and Name
 The name of the library containing the component

Library to be Linked
 The link specification that you use to link with this library

Header File The header file for the component, specified relative to */usr/include*

Other Required Headers
 Any other header files required to use the component, specified relative to */usr/include*. If the component requires multiple header files, separate the file names with spaces or commas.

Palette The name of the palette on which this component should appear

Icon Name
 The name that should appear on the palette. For long names, you can use an underscore to separate words. RapidApp converts the underscore to a newline when displaying the component. RapidApp automatically inserts underscores in mixed-case names.

Desktop Icon Drop List
 It's possible to set various resources of an interface element or class by dragging items from the desktop onto the element or class. For example, you can set the `labelPixmap` resource of a button by dragging a pixmap file onto the button. You can set the `fileName` resource of a scene viewer by dragging an inventor file onto the scene viewer.

To give this capability to a user-defined component, provide a list of file types and the corresponding "resource." The file type must be a name recognized by the SGI Indigo Magic desktop. The resource must be the name of a resource which corresponds to a method that accepts a filename. File types and resources are separated by colons, and pairs are separated by commas. For example:

```
TiffImageFile:setImageFile, XPMPixmapFile:setPixmap
```

Some components take advantage of this feature to make it easier to manipulate the component in RapidApp. For others, it's ignored. This feature does not affect programs built with the component.

3. Install or save the component information:
 - If you want to install the component on your local system so that RapidApp can use it, click the *Install Locally*. RapidApp saves the information in several files in your personal *\$HOME/.rapidappdir* directory. RapidApp also saves the files in the current directory and updates the files used by Software Packager so that the installable image you create contains the files as well. "Files Generated When Loading Components on a Palette" on page 134 describes these files in more detail.
 - If you want only to save the component information to the current directory, but don't want to install the files in your personal *\$HOME/.rapidappdir* directory, click the *Save/Generate* button. This creates the files you will need to create in installation package you can install on other machines.
4. Repeat steps 2 and 3 for any additional components you want to load onto RapidApp palettes. You can select new classes from the list of available classes, which can be displayed by selecting the *Browse...* menu item.
5. Click the *Close* button when you are finished specifying components to load.
6. Quit and restart RapidApp. The components you installed should now appear on the palettes you specified.

Note: Clicking the *Install* button saves the component information in several files in your personal *\$HOME/.rapidappdir* directory, which RapidApp reads when it starts. Because this information is stored in your home directory, it affects RapidApp only when you use RapidApp from your account. If you start RapidApp from another account on your system, the components don't appear. To install components so that they appear globally (that is, when you start RapidApp from any account on your system), you must create an installable image and load that image on your system, as described in "Creating an Installable Image" on page 136.

Example of Loading a Component Onto a RapidApp Palette

As an example, consider the case where you want to load the **Calculator** component onto a RapidApp palette. To do so, create the *libCalc* library as described in “Example of Creating a Component Library” on page 127, fill out the RapidApp Component Importer dialog as shown in Figure 8-2, and click the *Install* button. The next time you start RapidApp, it contains a MyComponents palette with a component on it that reads “My Simple Calculator.”

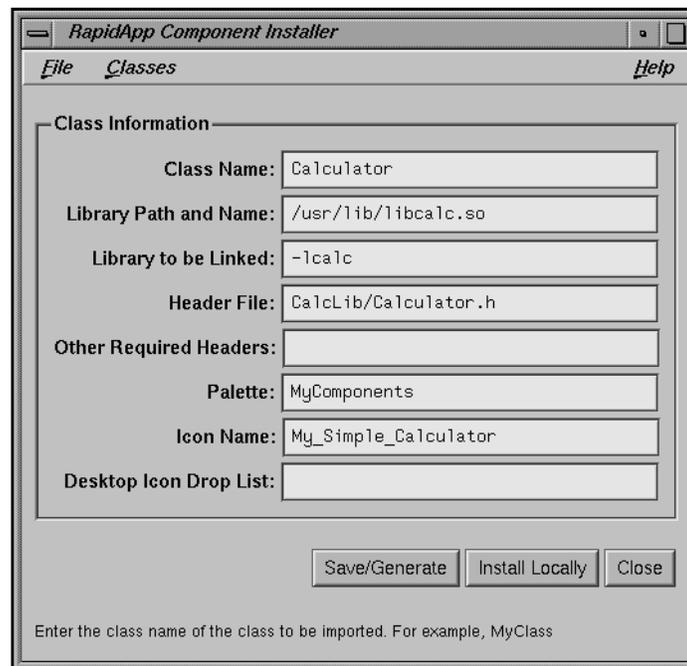


Figure 8-2 Example of Specifying a Component to Load on a RapidApp Palette

You can now use the Calculator component just like any built-in interface element. If you place a Calculator component in your interface and then switch to play mode, the Calculator component actually adds numbers. Furthermore, if you create an application using the Calculator, RapidApp doesn’t generate code for the Calculator component. Instead it links with the *libcalc.so* library and uses the component found in that library.

Files Generated When Loading Components on a Palette

RapidApp generates several component description files when you use the RapidApp Component Importer dialog to load components into RapidApp. In the following descriptions, *<ClassName>* represents the name of the component you are loading:

<ClassName>.item

A file that describes an item on a palette. The item file specifies information about the component including its name, its palette's name, and the pixmap used for the icon. RapidApp places this file in the *\$HOME/.rapidappdir/items* directory when you click the *Install* button in the RapidApp Component Importer dialog.

<ClassName>.col

A "collection file" that describes the component in more detail and sets some basic resources, such as the initial default size. This file also contains much of the information provided in the RapidApp Component Importer dialog, including the name of the library, the header file, and so on. RapidApp uses this information to load the component from the shared library when you create an instance of the component in an interface, and also uses the information for code generation. The item file tells RapidApp which collection file to load. RapidApp places this file in the *\$HOME/.rapidappdir/collections* directory when you click the *Install* button in the RapidApp Component Importer dialog.

<ClassName>.pix

The pixmap that appears on the palette for this component. The item file indicates the name of the pixmap file. RapidApp provides a default pixmap for the component in Xpm format. You can edit this file to provide a unique icon for your component. RapidApp places this file in the *\$HOME/.rapidappdir/pixmaps* directory when you click the *Install* button in the RapidApp Component Importer dialog.

`<ClassName>.wml`

A file that describes the Calculator component using a “Widget Metal Language” (wml). RapidApp requires this information for internal use; you shouldn’t need to modify this file for any reason. RapidApp places this file in the `$HOME/.rapidappdir/wml` directory when you click the *Install* button in the RapidApp Component Importer dialog.

`<ClassName.xres>`

A file that contains X-resource style descriptions of the help text RapidApp should display when the pointer moves over the component’s icon, or over resources supported by the component.

On startup, RapidApp reads the contents of `$HOME/.rapidappdir/items` and uses the information in each item file that it finds to load the other associated files.

Note: Clicking the *Install* button in the RapidApp Component Importer dialog saves the component description files in the appropriate subdirectories of your personal `$HOME/.rapidappdir` directory. Because this information is stored in your home directory, it affects RapidApp only when you use RapidApp from your account. If you start RapidApp from another account on your system, the components don’t appear. To install components so that they appear globally (that is, when you start RapidApp from any account on your system), you must create an installable image and load that image on your system, as described in “Creating an Installable Image.”

Deleting a Component from a Custom RapidApp Palette

You can delete a component from a personal custom palette by choosing the class to be removed from the Browser list and selecting the *Remove Local* item in the Classes menu.

You can also remove a component from your personal palette by deleting all of the component description files in the `$HOME/.rapidappdir` configuration directory. For example, to delete the component **MyComponent**, enter the following

```
% cd ~/.rapidappdir
% rm */MyComponent.*
```

To delete component that you installed globally from an installable image, simply use Software Manager to remove that installable image from your system.

Creating an Installable Image

The procedures described in “Loading Components Into RapidApp” allow you to load components onto RapidApp palettes on your system. However, you might also want to distribute these components to others. To do so, you must create an installable image of the library and the component description files:

1. Create a component library as described in “Build a Component Library” on page 125.
2. Install the component library on your system as described in “Install a Component Library” on page 127.
3. Use the RapidApp Component Importer dialog to save (or install and save) component description files for each component you want to add to a RapidApp palette, as described in “Loading Components Into RapidApp” on page 129. RapidApp automatically updates the files used by Software Packager so that the installable image you create contains the component description files as well.
4. Either use Software Packager to generate an installable image for distribution or simply enter `make images` in a shell window in your project directory. (For complete instructions for using Software Packager, consult the *Software Packager User's Guide*; Chapter 1, “Packaging Software for Installation: An Overview.”) The installable image places the component description files in `/usr/lib/RapidApp`, the same location RapidApp places its built-in components. After installing the libraries, the components automatically appear on their appropriate palettes whenever you start RapidApp.

Adding Resources to Components

When you load a component onto a RapidApp palette, you can create instances of the component and change the geometry of those instances. By default, you can't alter any of the other characteristics of the component; there are no built-in resources that you can change to affect the display and behavior of the component. However, you can add resources to your component that appear in the RapidApp resource editor. You can then select an instance of a component and change those resources to modify the appearance or behavior of the component.

In a component, resources correspond to C++ member functions. There are a few restrictions on the type and style of these member functions:

- Each member function must have a void return type
- The member function must not be virtual
- A member function may take exactly one argument, which must have one of the following types:
 - `const char *`: any string
 - `int`: An integer value
 - `float`: a floating point value
 - `Boolean`: a boolean value
 - An enum: an enumerated value, which must be zero-based.

A few other types are also possible, for special purposes. See “Arguments and Argument Types” on page 141 for details.

To illustrate adding resources to a component, first create a simple component called **LabeledText** to which you’ll add a couple of resources. The component consists of a label and a text field in a RowColumn container. Then add two resources to set the text shown in the label and to toggle the text field between editable and read-only mode.

To create the **LabeledText** component and add the desired resources:

1. Set up the application options:
 - From the File menu, choose “Preferences” to display the Preferences dialog.
 - In the Project card, enter `liblbt` in the Library name text field.
 - Enter `LibText` in the Library headers text field.
2. Create the component interface:
 - Create a RowColumn container. Set its **orientation** resource to `XmHORIZONTAL`.
 - Add a Label control to the RowColumn container.
 - Add a Text Field control to the RowColumn container.

- Select the RowColumn container and then select “Make Class” from the Classes menu. In the Make Class dialog, enter “LabeledText” as the class name. Your interface should look similar to that shown in Figure 8-3.

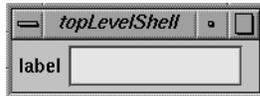


Figure 8-3 LabeledText Component

3. Save the interface and generate code:
 - Select “Save” from the File menu to save the interface.
 - Select “Generate C++” from the Project menu to generate the files for the component.
4. Create public member functions of the class to set the resources. The functions must have void return values and accept a single argument, whose type may be String, Boolean, integer, float, filename, or an enumerated type.

- Add declarations for two public member functions in the file *LabeledText.h*:

```
void setLabel(const char *);
void setReadOnly(Boolean);
```

- Add the source for the member functions in the file *LabeledText.C*:

```
void LabeledText::setLabel(const char * str)
{
    XmString xmstr = XmStringCreateLocalized( (String) str);
    XtVaSetValues(_label, XmNlabelString, xmstr, NULL);
}

void LabeledText::setReadOnly(Boolean readonly)
{
    XtVaSetValues(_textfield, XmNeditable, !readonly, NULL);
}
```

5. Add entries to the class's *interface map* data member. The first entry in each line indicates the resource that will appear in RapidApp for the user to set interactively. The second is a string that is the name of the member function. The final argument must be one of `XmRString`, `XmRBoolean`, `XmRInt`, `XmRFloat`, `VkRFilename`, `VkRNoArg`, `XmRCallback`, or an enumeration, and indicates the type of the single argument supported by the member function associated with this item.

- Find the section of code in *LabeledText.C* that looks like:

```
static VkCallbackObject::InterfaceMap map[] = {
    // { "resourceName", "setAttribute", XmRString},
    { NULL }, // MUST be NULL terminated
};
```

Note: On 5.3 systems, this structure will be declared in the code as simply "InterfaceMap".

- Follow the example shown and add each of the member functions to the table:

```
static VkCallbackObject::InterfaceMap map[] = {
    {"label", "setLabel", XmRString},
    {"readonly", "setReadOnly", XmRBoolean},
    { NULL }, // MUST be NULL terminated
};
```

6. Save *LabeledText.h* and *LabeledText.C*.
7. Generate the library by selecting "Build Application" from the Project menu.

You can now install the library as described in “Install a Component Library” on page 127, and load the library as described in “Loading Components Into RapidApp” on page 129. When you restart RapidApp, the Labeled Text component should appear on the appropriate palette. You can now add this component to your interface as you would any other interface element. When you do so, the result should be similar to that shown in Figure 8-4.

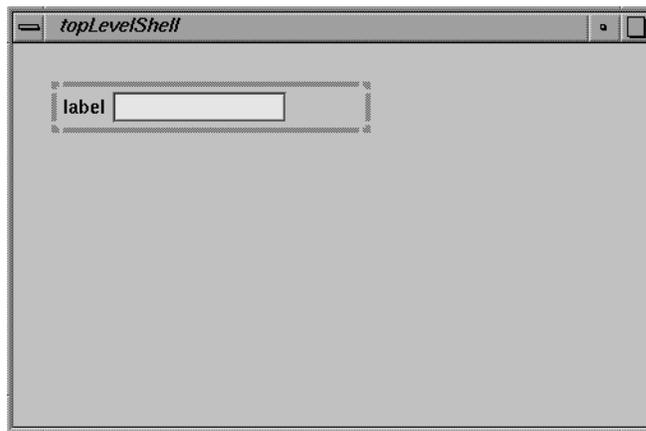


Figure 8-4 Using the LabeledText Component

Also, when you have the LabeledText component selected, the RapidApp resource editor should display the resources you added, as shown in Figure 8-5.

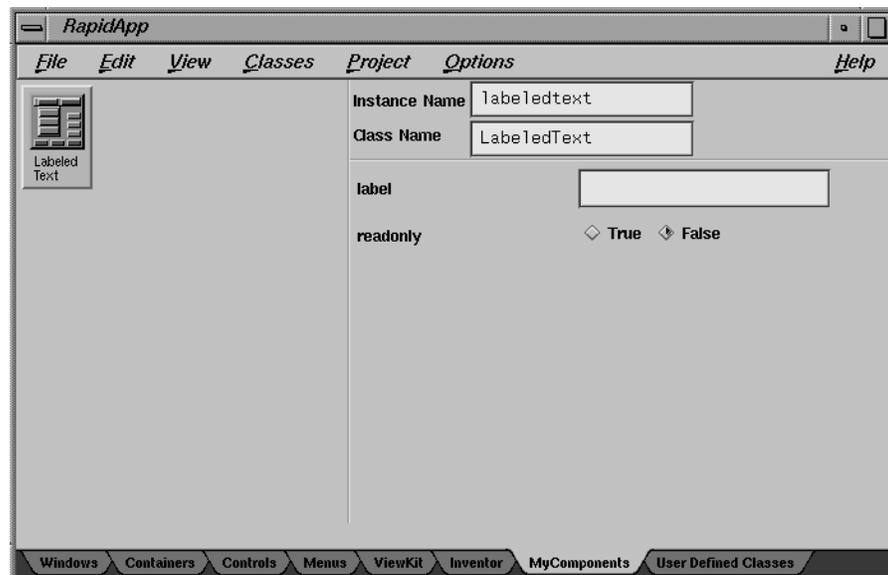


Figure 8-5 LabeledText Resources

Arguments and Argument Types

Member functions made available as resources in RapidApp must conform to a few strict requirements. Functions must be public and have a void return type. They must not be virtual, and they must take a single argument. This single argument can be of one of the following types:

- `const char *`. You can designate this argument type in the interface map structure using `XmRString`.
- `Boolean`. You can designate this argument type in the interface map structure using `XmRBoolean`.
- `int`. You can designate this argument type in the interface map structure using `XmRInt`.
- `float`. You can designate this argument type in the interface map structure using `XmRFloat`.

- A `const char *` that represents a file name. You can designate this argument type in the interface map structure using `VkRFilename` on IRIX6.2 or later systems. On earlier systems, use the string `"Filename"`. RapidApp will provide a file input field, with a drop pocket for this type of resource.
- An enumeration. You can designate this argument type in the interface map structure using a string of the form:

```
"Enumeration:Qualifier:Type: VALUE1, VALUE2, VALUE3"
```

Enumeration is a keyword that indicates that this is a description of an enumerated type. This word must be followed by a colon. The next two fields must indicate the type of the enumeration. If this enumeration is declared as part of a class, then the `Qualifier` field is the name of that class, while the `Type` is the name of the enumeration. If the enum is not part of a class, the `Qualifier` field can be blank, or set to the word `"Global"`

- Functions may also take no argument, indicated by the symbol `VkRNoArg`, on IRIX 6.2 or later. On IRIX 5.3 and other earlier systems, use the string `"NoArg"`. However, member functions registered in this way are not currently used within RapidApp.

Support for Callbacks

Member functions presented as resources in RapidApp, as described above, allow an object to be manipulated from the outside. However, it is also useful for objects to have a form of output, and to be able to communicate that something has changed. The ViewKit library, on which the code generated by RapidApp is based, supports an easy way to add callbacks to C++ classes.

To add a callback to a class, simply define a constant string as a public member of the class. For example:

```
class MyClass : public VkComponent {
const char *const myCallback;
// ...
};
```

Then define the string in the source file:

```
const char *const MyClass::myCallback = "myCallback";
```

Once defined, this class can invoke callbacks by simply calling:

```
callCallback(myCallback, (void *) data);
```

Other classes can register callbacks using `VkAddCallbackMethod()`, or `VkAddCallbackFunction()`.

To make this callback available in `RapidApp`, simply add an entry to the `InterfaceMap` structure, like this:

```
static VkCallbackObject::InterfaceMap map[] = {  
    {"myCallback", NULL, XmRCallback},  
    { NULL }, // MUST be NULL terminated  
};
```

The first item is the name of the callback. The second parameter can be `NULL` if this callback is defined by this class, or it can be set to the class name that defines the string, if the class is different. This might be true if the callback is inherited, for example. The final parameter must be `XmRCallback`, to indicate that this is a callback.

When this component is loaded by `RapidApp`, the callback will be made available as a resource, just like callbacks for Motif widgets.

Load Non-RapidApp Components

This section includes the following:

- “Introduction”
- “Requirements for Loading Components Not Created With RapidApp”
- “Testing Components”
- “Adding Custom Base Classes to RapidApp”

Introduction

In addition to loading components created in `RapidApp`, you can load other C++ components as long as they follow certain requirements. This section describes those requirements, and provides special instructions for loading components for which you don't have the source code (for example, components in third-party libraries).

Caution: Whether you create classes using RapidApp, or write them by hand, it is important to guard against bugs that could affect the environment in which the component will be executed. For example, any memory corruption problem in a component loaded into RapidApp could ultimately cause RapidApp itself to crash. You should test all components with a test driver and with a memory checker to avoid memory corruption problems. You should then use the Component Tester application provided with RapidApp to test the component, as discussed in “Testing Components” on page 147.

Requirements for Loading Components Not Created With RapidApp

There are certain general requirements and guidelines that components must follow for you to use them with RapidApp successfully:

- They must be based on IRIS ViewKit and IRIS IM. Specifically, the component must be derived from the IRIS ViewKit **VkComponent** base class, described in Chapter 2, “Components,” of the *IRIS ViewKit Programmer’s Guide*.
- They should be self-contained and not require complex initialization or other external components as part of their visible interface to operate. It is permissible for a component to contain or create other components as long as they are encapsulated. However, RapidApp has no way to deal with a component that requires an instance of another object to be passed to it, for example.
- They shouldn’t cause side effects outside the component that could affect the environment in which it is instantiated. For example, a component shouldn’t call **exit()** in the event of an error. Because the component is instantiated within the same address space and process as RapidApp, a component that calls **exit()** in response to invalid input also causes RapidApp to exit. Similarly, components should not install signal handlers, which may interfere with other components in RapidApp’s address space, or RapidApp itself.
- They should be able to be destroyed, and re-instantiated at any time. RapidApp often destroys and re-creates interface elements as part of its manipulations.
- They should behave properly when resized. Using RapidApp, the user can force a component to have any size or shape. If a component is useful only at a given size or shape, it is less useful to users.
- They should be able to be instantiated as many times as necessary, including having multiple instances in existence at once.

Additionally, all components that you want to load into RapidApp must follow a particular API. Most of these requirements are simply a restatement of the **VkComponent** protocol, but there are some features that are RapidApp requirements:

- Derive all components either directly or indirectly from **VkComponent**, and follow that class's API, as described in Chapter 2, "Components," of the *IRIS ViewKit Programmer's Guide*. In general, avoid multiple inheritance. If you do use multiple inheritance, you must list the **VkComponent** class first in the base class list.

- The component must have a constructor with the form:

```
Component::Component(const char *name, Widget parent)
```

If the constructor supports additional arguments, they must have default values and not be required for proper operation of the component.

Instantiating the component should create any widgets required by the component. The widgets must form a single-rooted subtree. The root of the widget subtree is referred to as the *base widget* of the component. Components typically use a container widget as the root of the subtree; all other widgets are descendants of this widget. The constructor should manage all widgets except the base widget, which should be left unmanaged.

See "Component Constructors" in Chapter 2 of the *IRIS ViewKit Programmer's Guide* for more information on component constructors.

- Every component should assign the widget that serves as the root of the component's widget hierarchy to the data member `_baseWidget`, which is inherited from **VkComponent**. RapidApp expects the **baseWidget()** member function, which is also inherited from **VkComponent**, to return this widget.
- Every component must have a **show()** and a **hide()** member function, which may be inherited from **VkComponent**. **show()** must make the component visible, while **hide()** is expected to remove the component from the screen. See "Displaying and Hiding Components" in Chapter 2 of the *IRIS ViewKit Programmer's Guide* for more information on the **show()** and **hide()** member functions.
- You must provide two static member functions, **Create<ClassName>()** and **Register<ClassName>Resources()**, where `<ClassName>` is the name of the class. **Create<ClassName>()** creates and displays an instance of a component. **Register<ClassName>Resources()** registers with RapidApp any component resources that you can set using the RapidApp resource editor. For example, the following code is generated by RapidApp for the **Calculator** class:

```
////////////////////////////////////
// C-callable creation function, for importing class into rapidapp
////////////////////////////////////
VkComponent * Calculator::CreateCalculator( const char *name, Widget
parent )
{
    VkComponent *obj = new Calculator(name, parent);

    return (obj);
}

////////////////////////////////////
// Function for importing this class into rapidapp
////////////////////////////////////

void* Calculator::RegisterCalculatorInterface()
{
    // This structure registers information about this class
    // that allows RapidApp to create and manipulate an instance.
    // Each entry provides a resource name that will appear in the
    // resource manager palette when an instance of this class is
    // selected, the name of the member function as a string,
    // and the type of the single argument to this function.
    // All functions must have the form
    //
    // void memberFunction(Type);
    //
    // where "Type" is one of:
    // const char * (Use XmRString)
    // Boolean (Use XmRBoolean)
    // int (Use XmRInt)
    // float (Use XmRFloat)
    // No argument (Use VkRNoArg)
    // A filename (Use VkRFilename)
    // An enumeration (Use "Enumeration:ClassName:Type:VALUE1,
VALUE2, VALUE3")
    // A callback (Use XmRCallback)

    static VkCallbackObject::InterfaceMap map[] = {
        // { "resourceName", "setAttribute", XmRString},
        { NULL }, // MUST be NULL terminated
    };

    return map;
}
```

Note: On systems earlier than IRIX 6.2, the type of the InterfaceMap structure is not defined by the VkCallbackObject class. You can see the correct structure in the Component.5.3 example, distributed with RapidApp.

Testing Components

Before loading a component into RapidApp, you should use the Component Tester application provided with RapidApp to test the component to make sure that there are no unexpected side effects that would affect the operation of RapidApp. To run the Component Tester, enter:

```
% comptester
```

The Component Tester appears as shown in Figure 8-6.

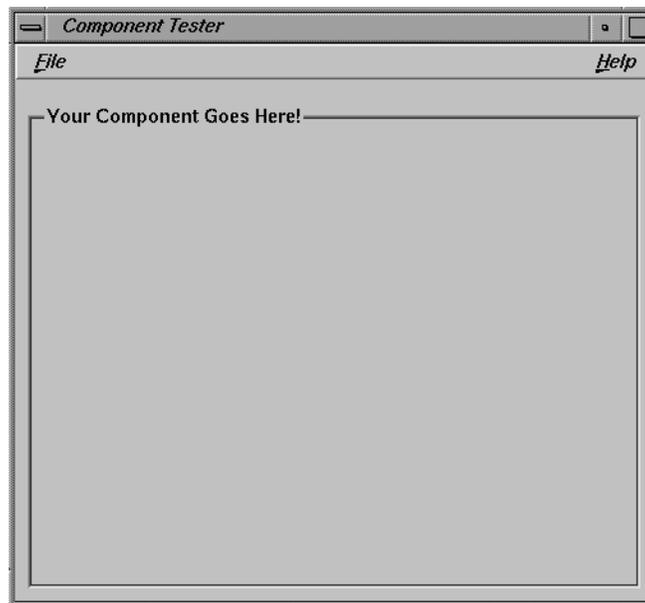


Figure 8-6 The Component Tester

To load a component into the Component Tester, select “Load Component” from the File menu. The Component Tester displays the Load Component dialog shown in Figure 8-7.

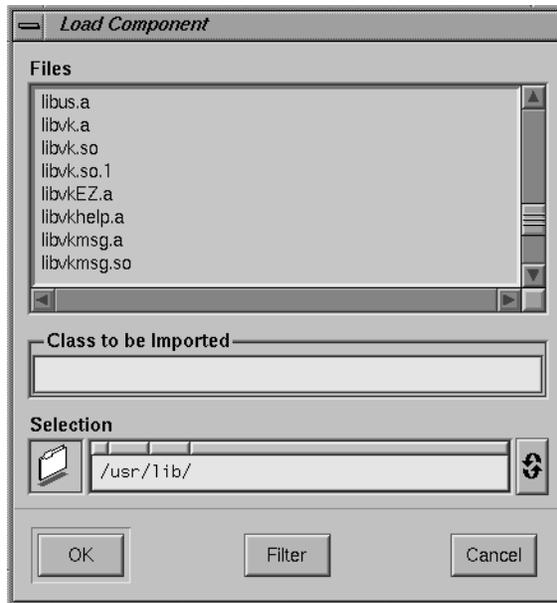


Figure 8-7 The Component Tester Load Component Dialog

Specify the class name of the component in the “Class to be Imported” field, and the DSO library in the “Selection” field, then click to *OK* button to load the component.

The Component Tester displays the components you load and allows you to manipulate them. If your component contains any component resources, the Component Tester displays them as items in an Operations menu. You can then select these menu items to change the resource values.

Completely exercise all features of your component. If there are no problems, you can safely use this component with RapidApp. If there are problems, correct them before trying to use the component with RapidApp. Source code for the Component Tester is included in `/usr/share/src/RapidApp/ComponentTester/`. You can compile the application using the debug libraries if you need further information while debugging your component.

Adding Custom Base Classes to RapidApp

When you declare a widget or collection of widgets as a class, you are offered a choice of base classes from which to derive the new class. If the user interface element is a `VkWindow` or `SimpleWindow`, you are offered the corresponding ViewKit window classes. Otherwise, the choices are `VkComponent`, or `VkMsgComponent`, which are also ViewKit classes.

In some cases, you may wish to provide your own classes from which to derived components created with RapidApp. You can do so, subject to the following restrictions:

- The class must be a subclass, direct or indirect, of `VkComponent`, `VkWindow`, or `VkSimpleWindow`.
- Only window classes can be derived from `VkWindow` or `VkSimpleWindow`, or user-added subclasses. All other classes must derived from `VkComponent`, or a user-added subclass of `VkComponent`.
- Your class must not have any user interface.
- The class must present the same derived class protocol as `VkComponent`, (or in the case of window classes, `VkWindow` or `VkSimpleWindow`).

The most common use of this feature is to create a class that supports some additional programmatic protocol that you would like other classes to inherit.

An Example Base Class

For example, consider a new class, `ImageComponent`, that adds a public member function to `VkComponent`, `void loadImageFile()`. The class could be declared as follows:

```
#include <Vk/VkComponent.h>
class ImageComponent : public VkComponent {
public:
    void loadImageFile(char *filename);
};
```

To allow this class to be chosen as the base class of new components created in RapidApp, the class must be registered with RapidApp. This requires the following steps:

1. Create a file with the same name as your class. In the above example, the file would be named ImageComponent.
2. Add lines to this file to describe the header file and location for this class and also the library in which this class can be found. The format of this file is the same as an X resource file. For example, if the declaration of this class is found in a file ImageComponent.h, which is normally located in /usr/include/Image, and the class is in a library named libimage.so, you could write this file as follows:

```
! RapidApp registration file for the ImageComponent class
*ImageComponent*headerFile: <Image/ImageComponent.h>
*ImageComponent*library: -limage
```

3. Install this file in one of two locations.
 - If you are just testing, or using this class for your own personal use, you can place this file in

```
$HOME/.rapidappdir/BaseClasses/VkComponent/ImageComponent.
```

- If you wish to share this class with others, or make it a more permanent part of your development environment, you can install the file in

```
/usr/lib/RapidApp/BaseClasses/VkComponent/ImageComponent
```

You can describe new subclasses of VkWindow or VkSimpleWindow to RapidApp in the same way. The restrictions are the same, as are the installation steps. However, subclasses of VkWindow must be placed in *\$HOME/.rapidappdir/BaseClasses/VkWindow*, or */usr/lib/RapidApp/BaseClasses/VkWindow*

Subclasses of VkSimpleWindow must be placed in *\$HOME/.rapidappdir/BaseClasses/VkSimpleWindow*, or */usr/lib/RapidApp/BaseClasses/VkSimpleWindow*.

Example Applications

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 9-1.

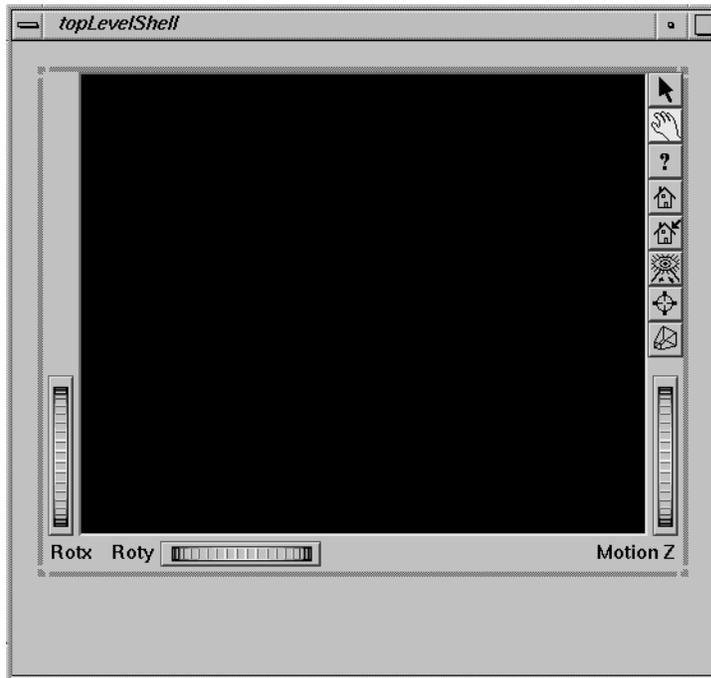


Figure 9-1 Adding an Examiner Viewer

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 9-2 shows the completed interface.

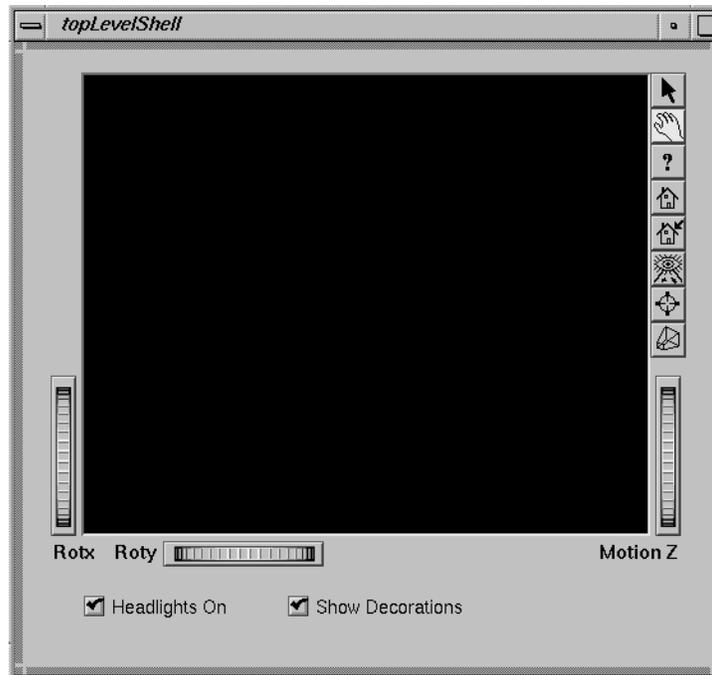


Figure 9-2 The Completed Open Inventor Component

6. From the File menu, choose “Preferences.”
7. In the Project card of the Preferences dialog, change the Application name to “cone.”
8. 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 Project card on the Preferences dialog, RapidApp creates the directory for you automatically if it doesn’t already exist.
9. Select “Generate C++” from the Project menu to generate code. Then selection “Build Application” from the Project menu to build the application.

10. 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 calls ConePanelUI(parent, name)
    // which calls ConePanelUI::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
```

- 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 9-3.

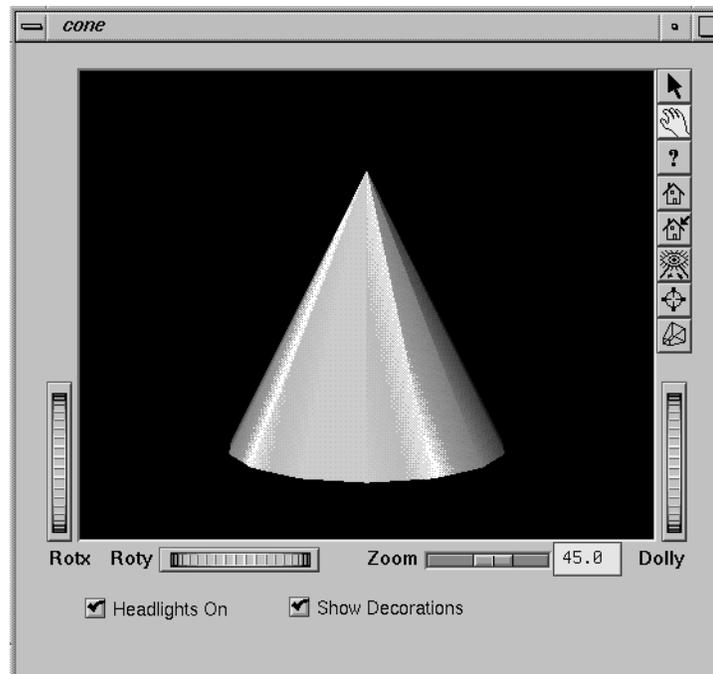


Figure 9-3 The Open Inventor Interface Displaying a Scene

11. 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 )
{
    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:

} // End ConePanel::headlight()
```

- 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);
} // End ConePanel::headlight()
```

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 )
{
    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:

} // End ConePanel::toggleDecorations()
```

- 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);
} // End ConePanel::toggleDecorations()
```

- 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 them 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 window 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.

Dialogs/PhoneDialog

A program that shows how to create dialogs as subclasses of the IRIS ViewKit **VkDialogManager** class, and connect the dialog to the rest of the program.

Dialogs/Question

A program that shows how to create dialogs as subclasses of the ViewKit **VkQuestionDialog** class, and connect the dialog to the rest of the program.

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/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.

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.

AppClass A trivial example that creates its own subclass of **VkApp**.

Component An example of creating a C++ class that can be imported back onto RapidApp's palettes. This component is a simple combination of a label and a text input field. It has member functions that support changing the label, setting the text field to read only, and setting a "mask" that determines what input is allowed. This component can be added to the RapidApp palettes using "Install Class" from the RapidApp Classes menu, or by creating an installable image from the files in this directory.

ComponentTester
If you try to load an untested component into RapidApp, it is easy to crash RapidApp because the component is executed in RapidApp's address space. This example provides a simple program that is able to load a component and test it. Menu items allow you to set "resources" for an object. Once you can successfully load a component into this test program, you should have no trouble using it in RapidApp. Because you have source, you can run this program under a debugger if necessary to isolate any problems. See "Testing Components" on page 147 for more information on the Component Tester.

The following examples require the Inventor Development option:

HelloCone An example of using RapidApp and IRIS ViewKit with Open Inventor. This example is based on the HelloCone example in *The Inventor Mentor*.

IvViewer Another IRIS ViewKit/Open Inventor example. This program allows you to open an Open Inventor data file and view it.

MiniSim A very simple "flight simulator", that shows a plane in flight, and allows you to control some aspects of its movement. Based on an Inventor scene viewer, this program demonstrates how to connect an Inventor database and user interface controls.

RapidApp Reference

Reference

This appendix describes in the function of each window, menu, widget, and display in the RapidApp's graphical user interface (GUI). RapidApp consists of several palettes, each one containing several user interface elements. Each palette and its interface elements are described in detail in their own sections in this chapter:

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

Global Objects

This section describes RapidApp’s global objects—the objects that are common across all palettes. These objects are the menu bar items, the dialogs, and the palette tabs (see Figure A-1).

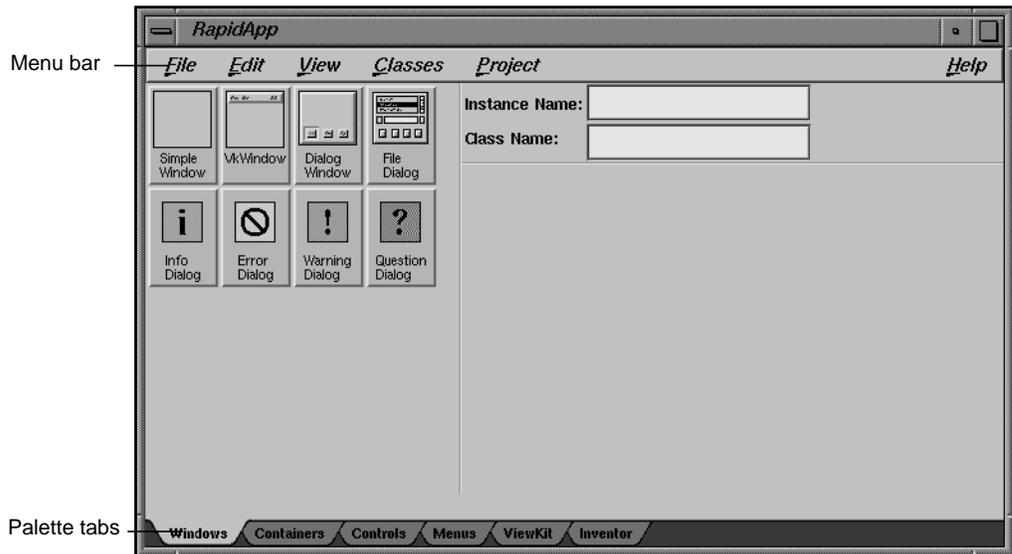


Figure A-1 RapidApp Main Window

File Menu

<i>Open...</i>	<i>Ctrl+O</i>
<i>Import...</i>	<i>Ctrl+I</i>
<i>New</i>	<i>Ctrl+N</i>
<i>Save</i>	<i>Ctrl+S</i>
<i>Save As...</i>	<i>Ctrl+A</i>
<i>Preferences...</i>	
<i>Exit</i>	<i>Ctrl+Q</i>

Figure A-2
File Menu

The File menu (see Figure A-2) allows you to open and save 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 one. 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 <i>save.uil</i> .
Save As	Displays the Save File dialog, which allows you to save your current session to a file with a filename of your choice.
Preferences	Displays the RapidApp Preferences dialog which contains the following cards: <ul style="list-style-type: none"> • Project—allows you to specify application file and class names and various application characteristics. • RapidApp—allows you to set preferences controlling RapidApp operation. • Code Style—allows you to specify various options that affect code generation. • Merge Options—allows you to specify how RapidApp merges code in various files. <p>The cards are discussed in the “RapidApp Preferences Dialog” section.</p>
Exit	Exits RapidApp.

Edit Menu

The Edit menu (see Figure A-3) supports cut, copy, and paste operations, as well as 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 places it 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 <i>doesn't</i> place the element on the clipboard.
Up/Left	In containers where the creation order of its child elements determines their position, moves the currently selected child one position up or left.

<i>Cu</i> t	<i>Ctrl+X</i>
<i>C</i> opy	<i>Ctrl+C</i>
<i>P</i> aste	<i>Ctrl+V</i>
<i>D</i> elete	<i>Delete</i>
<i>U</i> p/ <i>L</i> eft	<i>Ctrl+U</i>
<i>D</i> own/ <i>R</i> ight	<i>Ctrl+D</i>
<i>S</i> elect Parent	<i>Ctrl+P</i>
<i>N</i> atural Size	
<i>G</i> row Widget	<i>Ctrl+G</i>
<input type="checkbox"/> <i>S</i> how Menu	

Figure A-3
Edit Menu

- Down/Right In 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.
- Natural Size Resize the currently selected element to its default size. Note that this option has no effect if the element is a child of a container that controls its size (for example, a Row Column widget).
- 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 A-4
View Menu

The View menu (see Figure A-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.
- Color by Depth When toggled on, RapidApp displays nested containers in different colors to indicate their depth.
- Snap To Grid Submenu (see Figure A-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 the toggle is 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 any other container on which you drop them.



Figure A-5
Snap To Grid
Toggle



Figure A-6
Classes Menu

Classes Menu

The Classes menu (see Figure A-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 this toggle is 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.
- Install Class** Displays the RapidApp Component Installer dialog for loading a user-defined component onto a RapidApp palette. See “Loading Components Into RapidApp” on page 129 for more information.

Project Menu



Figure A-7
Project Menu

The Project menu (see Figure A-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.
- Build Application** Launches 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.
- 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.

RapidApp Preferences Dialog

The RapidApp Preferences Dialog provides several tabbed-cards that allow you to set options to determine how RapidApp behaves when working with your application. The following cards are included:

- "Project Card"—allows you to specify application file and class names and various application characteristics.
- "RapidApp Card"—allows you to set preferences controlling RapidApp operation.
- "Code Style Card"—allows you to specify various options that affect code generation.
- "Merge Options Card"—allows you to specify how RapidApp merges code in various files.

Project Card

The Project card (Figure A-8) 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.

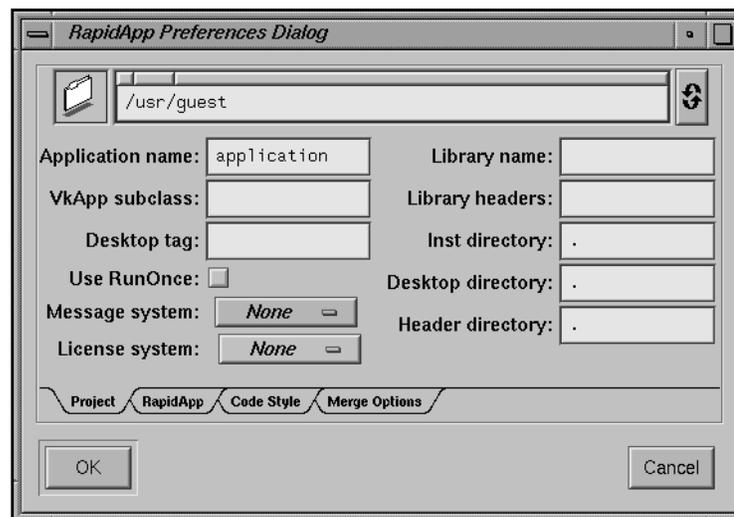


Figure A-8 The Project Card

The following options are available:

Directory Path (field at the top of the card)

Enter the directory in which the application source are placed when generated. If the directory does not exist, RapidApp posts a question asking if it should be created.

Application name

The name of the program to be created. Following X conventions, RapidApp automatically determines the application class from this setting by capitalizing the first letter of the application name.

Library name The name of a library. RapidApp generates a Makefile that creates libraries of all the classes you create.

VkApp subclass

The name of a class. RapidApp creates a subclass of **VkApp** using the subclass name you specify for the application instead of directly instantiating a **VkApp** object. This allows you to modify the **VkApp** subclass to handle application-specific needs (for example, parsing command-line options).

Library headers

The pathname, relative to */usr/include*, where RapidApp installs the header files for your user-defined component library.

Desktop tag

The name of a workspace tag to be used with the desktop.

Inst directory

The name of the directory where RapidApp puts the inst-able images for your application.

Use RunOnce

If set, the program uses the **VkRunOnce** facility, which ensures that only one instance of the application is running at any one time. See the **VkRunOnce(3Vk)** reference page for details.

Desktop directory

The directory in which to place all desktop support files.

Message system

If set to Tooltalk, the application supports basic ToolTalk™ communication using the ViewKit **VkMsg** facility.

Header directory

The name of the directory where your header files are saved. Typically a subdirectory of your project directory.

License system

If set to Net LS, RapidApp generates an application that includes the code to set up the application to use the NetLS™ license system. See the **VkNLS(3Vk)** reference pages for details.

If set to Flex LM, RapidApp generates an application that includes the code to set up the application to use the FlexLM™ license system. See the **VkFLM(3VK)** reference pages for details.

If set to None, RapidApp doesn't generate code to license the application.

RapidApp Card

The RapidApp card (Figure A-9) allows you to customize the behavior of RapidApp itself.

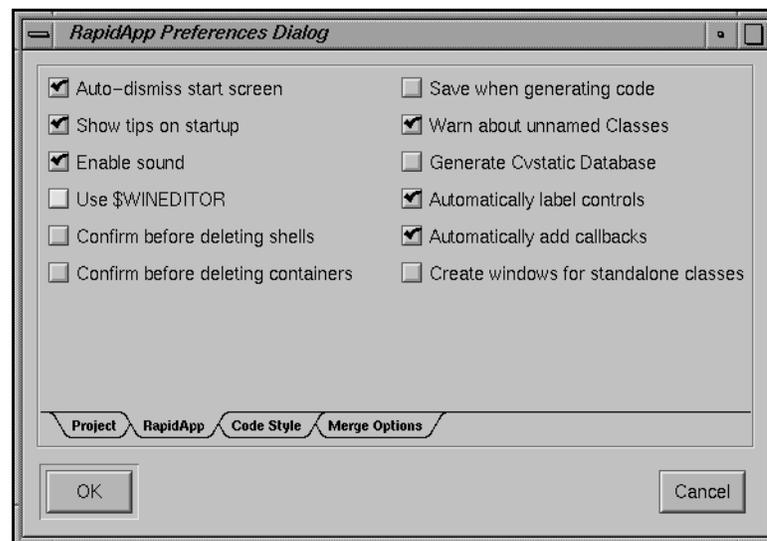


Figure A-9 The RapidApp Card

The following options are available:

Auto-dismiss start screen

If set, the start screen disappears as soon as the program is ready to run without the need to manually dismiss it.

Show tips on startup

If set, show a random tip on the start screen about how to use RapidApp.

Enable sound

If set, 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

If set, "Edit File" on the Project menu launches the window-based editor indicated by the WINEDITOR environment variable. If unset, RapidApp launches the Developer Magic Source View editor.

Confirm before deleting shells

If set, RapidApp posts a warning dialog if you attempt to dismiss a top-level user interface element using the Close option of the window manager menu.

Confirm before deleting containers

If set, RapidApp posts a warning dialog whenever you delete a container, to avoid accidental deletions of elements that might be hard to reconstruct.

Save when generating code

If set, RapidApp saves your interface automatically whenever you generate C++ code.

Warn about unnamed Classes

If set, RapidApp posts a warning before creating any classes automatically. RapidApp forces elements such as windows and dialogs to be classes and generates a name if you haven't provided one by declaring the element as a class. The warning offers you a chance to abort the creation and specify your own name.

Generate Cvstatic Database

If set, RapidApp automatically creates a static analysis database whenever you compile your application.

Automatically label controls

If set, RapidApp generates a labelString automatically the first time you name a label or label subclass (e.g. a button), unless you've already provided a labelString. The algorithm is to capitalize the first letter of the name and split at any capital letters. For example, RapidApp computes the labelString for a button named openFile to be "Open File."

Automatically add callbacks

If set, RapidApp generates a callback function automatically the first time you name a label or a label subclass (e.g. a button), unless you've already provided a callback. The algorithm is to capitalize the first letter of the name and add "do" to activateCallbacks and "set" to valueChanged callbacks. For example, RapidApp computes the callback for a button named openFile to be "doOpenFile()."

Create window for standalone classes

If True, any user interface element that is in a toplevel shell is 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.

Code Style Card

The Code Style card (Figure A-10) allows you to specify various options that affect code generation.

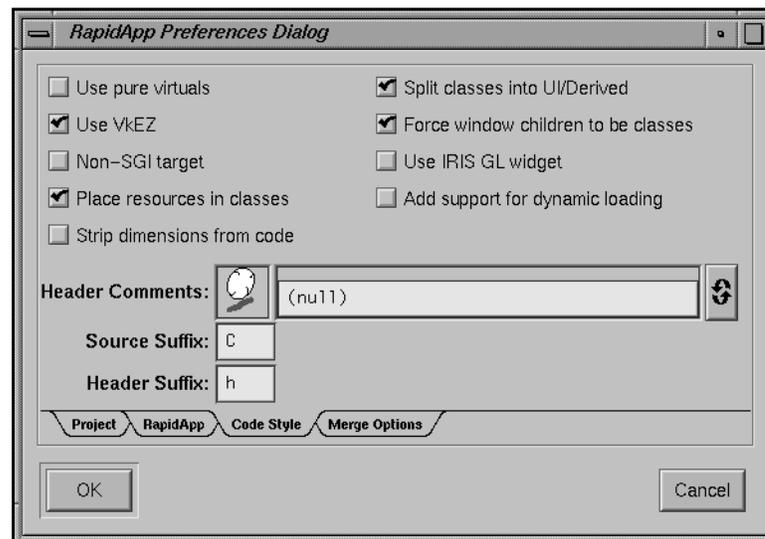


Figure A-10 The Code Style Card

The following options are available:

Use pure virtuals

If set, RapidApp generates pure virtual functions in base classes.

Use VkeZ

If set, the generated code includes the VkeZ headers and libraries.

Non-SGI target

If set, RapidApp generates a Makefile that doesn't use the macros found in */usr/include/make*. This provides a Makefile that you can use and modify to compile code for systems other than Silicon Graphics.

Place resources in classes

When True, RapidApp places default resources for classes in class data structures rather than an application resource file. This allows you to encapsulate the resources with their corresponding classes when creating class libraries.

Strip dimensions from code

Strip all width and height resources from code.

Split classes into UI/Derived

If set, RapidApp splits created classes into a base and a derived class. This can be overridden on a per-class basis through the Make Classes dialog.

Force window children to be classes

If set, windows, dialogs, and the VktabbedDeck force their single child to be a class. This can be overridden on a per-object basis.

Use IRIS GL widget

If set, the IRIS GL widget is used for the GLDrawing area container in place of OpenGL widget.

Add support for dynamic loading

If set, RapidApp allows you to create components that can be loaded back onto the RapidApp palette as first class objects. To support this, RapidApp adds two simple functions to each class, to allow them to be dynamically loaded. This can be overridden on a per-class basis through the Make Classes dialog.

Header Comments

A file name. The file is inserted at the top of each generated file. For example, this file can be used to supply a company standard header.

- Source Suffix** A suffix. The source suffix can be any valid suffix supported by the C++ compiler and Silicon Graphics standard *Makefiles*. The accepted file extensions are: *.c*, *.C*, *.cxx*, and *.c++*.
- Header Suffix** A suffix. The header suffix can be any valid suffix supported by the C++ compiler and Silicon Graphics standard *Makefiles*. The accepted file extension is *.h*.

Merge Options Card

The Merge Option card (Figure A-11) affects how RapidApp merges code in various files.

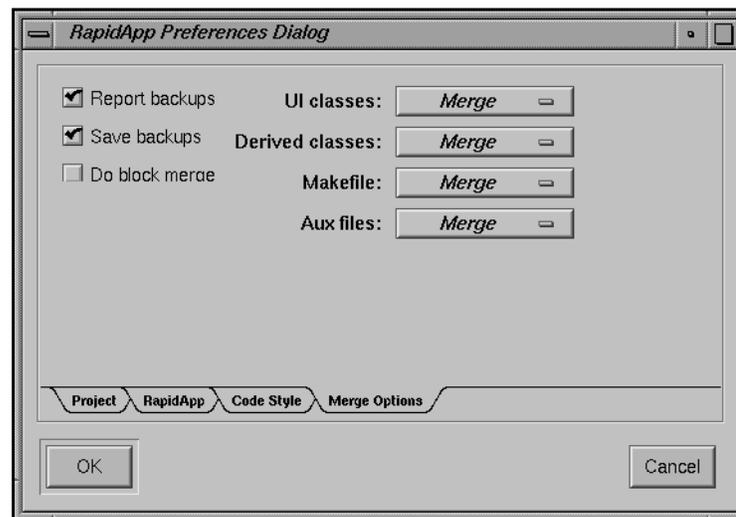


Figure A-11 The Merge Options Card

The following options are available:

Report backups

If set, RapidApp posts a dialog detailing the files that changed after it made a backup or merged code.

Save backups If set, RapidApp creates backups when any changes are made to code.

Do block merge

If set, RapidApp replaces all sections in the newly generated code that lie between editable code blocks with the code from the same section of the current file.

UI classes The base UI classes

Derived classes

The classes derived from the UI classes

Makefile The *Makefile* used to compile your application

Aux files Auxiliary files such as the application resources file, the desktop icon, the FTR file, and the files used by Software Packager to generate an installable image.

For each type of file, you can specify one of three methods for handling code generation:

Merge RapidApp attempts to merge any changes you have made to the existing file with the new file that it is generating. The merge process is described below. Typically, this is the best choice if you have edited any of the files (for example, to add functional code).

Don't Merge RapidApp writes all the new files with a *.rapidApp* extension. You must then merge the files by hand. You might want to specify this method if you make complex changes to files (for example, creating a highly customized *Makefile*) and want to be certain that your changes are preserved.

Overwrite RapidApp simply overwrites the existing file with the newly generated file. In this case, RapidApp always backs up the old file to the project's *.backup* directory even if you have turned off the backup feature by toggling off the Save backup copies option in the Code Generation Options dialog. This is the fastest method of the three and is usually appropriate for the UI classes unless you have made changes to those files by hand.

RapidApp Component Importer Dialog

The RapidApp Component Importer dialog allows you to load a user-defined component from a library onto a RapidApp palette. To load a class, you must fill in the following dialog fields:

Class Name The name of the component to import

Library Path and Name

 The complete pathname of the library containing the component

Library to be Linked

 The link specification that you use to link with this library

Header File The header file for the component, specified relative to */usr/include*

Other Required Headers

 Any other header files required to use the component, specified relative to */usr/include*. If the component requires multiple header files, separate the file names with spaces or commas.

Palette The name of the palette on which this component should appear

Icon Name The name that should appear on the palette. For long names, you can use an underscore to separate words. RapidApp converts the underscore to a newline when displaying the component.

Desktop Icon Drop List

 It's possible to set various resources of an interface element or class by dragging items from the desktop onto the element or class. For example, you can set the `labelPixmap` resource of a button by dragging a pixmap file onto the button. You can set the `fileName` resource of a scene viewer by dragging an inventor file onto the scene viewer.

 To give this capability to a user-defined component, provide a list of file types and the corresponding "resource." The file type must be a name recognized by the SGI Indigo Magic desktop. The resource must be the name of a resource which corresponds to a method that accepts a filename. File types and resources are separated by colons, and pairs are separated by commas. For example:

```
TiffImageFile:setImageFile, XPMPixmapFile:setPixmap
```

 Some components take advantage of this feature to make it easier to manipulate the component in RapidApp. For others, it's ignored. This feature does not affect programs built with the component.

If you want to install the component on your local system so that RapidApp can use it, click the *Install Locally* button. RapidApp saves the information in several files in your personal *\$HOME/.rapidappdir* directory. RapidApp also saves the files in the current directory and updates the files used by Software Packager so that the installable image you create will contain the files as well. “Loading a Component Onto a RapidApp Palette” on page 129 describes these files in more detail.

If you only want to save the component information to the current directory, but don’t want to install the files in your personal *\$HOME/.rapidappdir* directory, click the *Save/Generate* button.

Click the *Quit* button when you are finished specifying components to load.

See “Loading Components Into RapidApp” on page 129 for more information on loading components into RapidApp.

Palette Tabs

You access the RapidApp palettes through the palette tabs (see Figure A-12) at the bottom of the RapidApp window. Click a tab to display the corresponding palette.



Figure A-12 Palette Tabs

Keys and Shortcuts

This section describes the accelerator keys available in RapidApp.

Shift+F1	Displays context-sensitive help for the item you click.
Ctrl+O	Opens a file.
Ctrl+I	Imports a file.
Ctrl+N	Starts a new project, deleting all current elements.
Ctrl+S	Saves a file.
Ctrl+A	Saves a file as a new name.
Ctrl+P	Selects the parent of the currently selected element.

Shift+Ctrl+Left mouse	Selects 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 and places it on the clipboard.
Ctrl+C	Copies the currently selected element to the clipboard.
Ctrl+V	Pastes the contents of the clipboard.
Delete	Deletes the currently selected element without placing it on the clipboard.
Backspace	Deletes the currently selected element without placing it on the clipboard.
Ctrl+U	Repositions a widget inside a Row Column widget, moving it up, if the parent's orientation is vertical.
Ctrl+D	Repositions a widget inside a Row Column widget, moving it down, if the parent's orientation is vertical.
Ctrl+K	Toggles 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	Allows you to manipulate the currently selected element without accidentally selecting other elements.
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 accelerators 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 A-13) contains window interface elements.

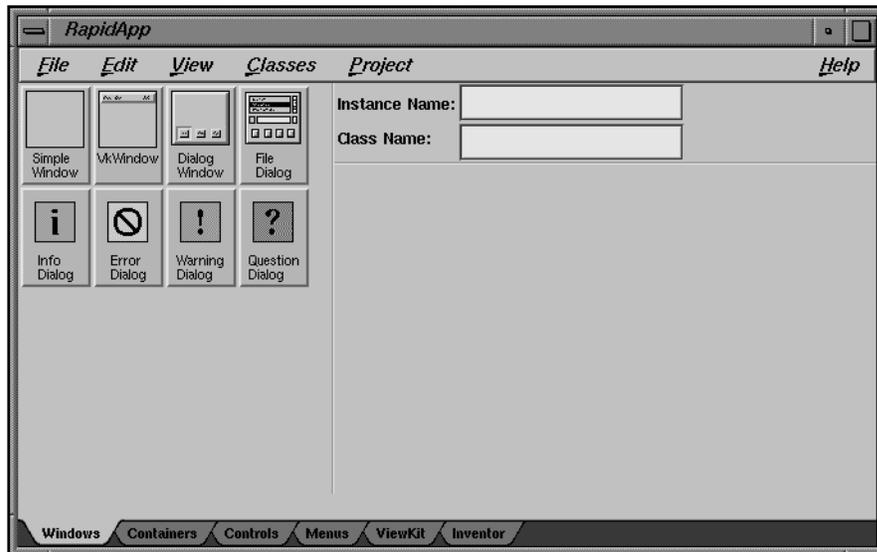


Figure A-13 Windows Palette

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

Simple Window

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:

autoRouteCallbacks

If True, then for each menu item in the window's menu bar for which you've defined an **activateCallback** function, RapidApp adds a member function of the same name to the window's child component or generated class.

coprimaryWindow

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

disableIconify If True, the user's ability to iconify the window is disabled.

disableWindowResize

If True, the user's ability to resize the window is disabled.

forceChildClass

If True, this window's child is implemented as a C++ class.

hideTitleBar If True, do not display this window's title.

hideWMBorder

If True, do not display this window's border. This option works when the title bar is also hidden.

showPopupHelp (version 6.2 only)

If True, show popup help for items that set the **popupMsgHelp** resource.

title

The title that appears in the window manager border.

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.

Dialog Window

The Dialog Window provides a top-level dialog window for constructing custom dialogs that conform to the API provided by the **VkDialogManager** class. To create a dialog, add a single container, 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(3Vk)** reference page for more information.

The Dialog Window has the following resources:

allowMultipleDialogs

If True, the dialog manager class can create multiple instances. If False, only one instance of this dialog can be created. Calling **post()** multiple times reuses the same dialog instance.

autoRouteCallbacks

If this resource is set to True, then for each menu item in the window's menu bar for which you've defined an **activateCallback** function, RapidApp adds a member function of the same name to the window's child component or generated class.

forceChildClass

If True, this window's child is implemented as a C++ class.

minimizeDialogs

If True, the dialog manager class attempts to minimize the number of dialog instances created. This resource is similar to the `allowMultipleDialogs` resource except that, in some cases, it may create multiple instance,

File Dialog

The File Dialog provides a file selection dialog for you to customize by adding an optional menu bar and/or adding a single child element. RapidApp creates the dialog as a subclass of **VkFileSelectionDialog**, which is a subclass of **VkDialogManager**. See the `VkFileSelectionDialog(3Vk)` reference page for more information.

You can post the file selection dialog programmatically by calling **post()**, **postBlocked()**, **postModal()**, or **postAndWait()**. Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. Consult the `VkDialogManager(3Vk)` reference page for more information on the functions provided by the **VkDialogManager** class.

Info Dialog

The Info Dialog provides an information dialog for you to customize by adding an optional menu bar and/or adding a single child element. RapidApp creates the dialog as a subclass of **VkInfoDialog**, which is a subclass of **VkDialogManager**. See the `VkInfoDialog(3Vk)` reference page for more information.

You can post the information dialog programmatically by calling **post()**, **postBlocked()**, **postModal()**, or **postAndWait()**. Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. Consult the `VkDialogManager(3Vk)` reference page for more information on the functions provided by the **VkDialogManager** class.

Error Dialog

The Error Dialog provides an error dialog for you to customize by adding an optional menu bar and/or adding a single child element. RapidApp creates the dialog as a subclass of **VkErrorDialog**, which is a subclass of **VkDialogManager**. See the **VkErrorDialog(3Vk)** reference page for more information.

You can post the error dialog programmatically by calling **post()**, **postBlocked()**, **postModal()**, or **postAndWait()**. Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. Consult the **VkDialogManager(3Vk)** reference page for more information on the functions provided by the **VkDialogManager** class.

Warning Dialog

The Warning Dialog provides a warning dialog for you to customize by adding an optional menu bar and/or adding a single child element. RapidApp creates the dialog as a subclass of **VkWarningDialog**, which is a subclass of **VkDialogManager**. See the **VkWarningDialog(3Vk)** reference page for more information.

You can post the warning dialog programmatically by calling **post()**, **postBlocked()**, **postModal()**, or **postAndWait()**. Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. Consult the **VkDialogManager(3Vk)** reference page for more information on the functions provided by the **VkDialogManager** class.

Question Dialog

The Question Dialog provides a question dialog for you to customize by adding an optional menu bar and/or adding a single child element. RapidApp creates the dialog as a subclass of **VkQuestionDialog**, which is a subclass of **VkDialogManager**. See the **VkQuestionDialog(3Vk)** reference page for more information.

You can post the question dialog programmatically by calling **post()**, **postBlocked()**, **postModal()**, or **postAndWait()**. Each function accepts arguments for setting the dialog message, callback functions for each button on the dialog, and other parameters. The **VkDialogManager** class also offers functions for setting the dialog's title, setting the labels for its buttons, programmatically dismissing the dialog, and other actions. Consult the **VkDialogManager(3Vk)** reference page for more information on the functions provided by the **VkDialogManager** class.

Containers Palette

The Containers palette (see Figure A-14) includes container interface elements such as bulletin boards and radio button boxes.

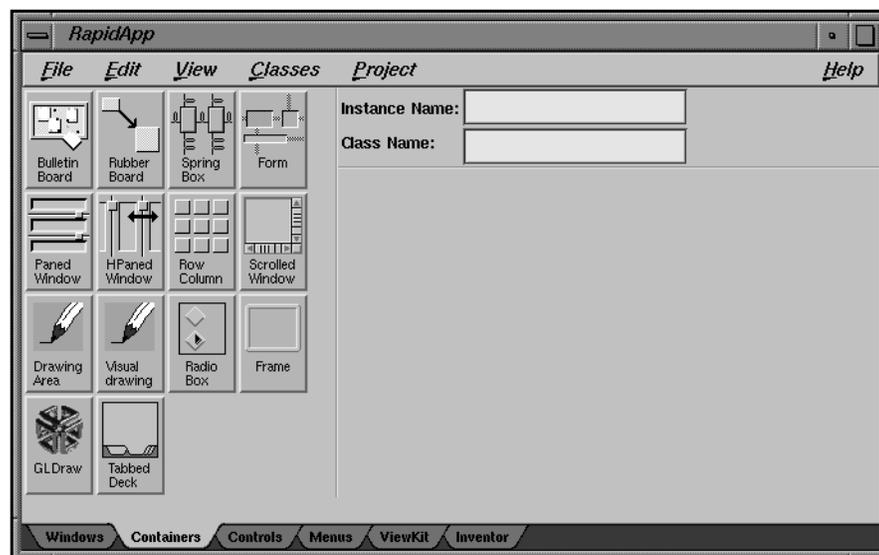


Figure A-14 Containers Palette

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

Bulletin Board

The Bulletin Board 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 Bulletin Board cannot be resized and do not respond to changes to individual interface elements.

Bulletin Board layouts are not appropriate for programs that are customized or internationalized. This container is most suitable for beginners and for quick prototypes.

Bulletin Board Resources

Following are the Bulletin Board resources:

XmNmarginHeight

Specifies the minimum spacing in pixels between the top or bottom edge of Bulletin Board and any child widget. You must be careful when positioning children using RapidApp, because the Bulletin Board enforces this margin only at creation time. The Bulletin Board 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 Bulletin Board 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 Bulletin Board and any child widget. The same restrictions apply as in the **XmNmarginHeight** resource.

Rubber Board

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

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

Rubber Board Resources

Following are the resources supported by the Rubber Board container:

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

Spring Box

The Spring Box 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 Spring Box allows layouts similar to those supported by the Form container, but is sometimes easier to set up and allows you to create some layouts that cannot be achieved with the Form container. For example, centering a column of interface elements is very easy to do with the Spring Box, but nearly impossible using the Form.

Each child of an Spring Box container has the following constraints associated with it:

- Each child has a “springiness” in both the vertical and horizontal directions 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 inRapidApp’s resource editor. Selecting any child displays its resources in the resource editor.

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

Spring Box Resources

Following are the Spring Box resources:

XmNmarginHeight

Specifies the minimum spacing in pixels between the top and bottom edges of the Spring Box and any child widget.

XmNmarginWidth

Specifies the minimum spacing in pixels between the left or right edge of the Spring Box and any child widget.

XmNminSpacing

Specifies the minimum spacing between the children of the Spring Box.

XmNorientation

The **XmNorientation** resource determines whether the Spring Box 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 new orientation.

XmNdefaultVerticalLayout, XmNdefaultHorizontalLayout

These resources provide a convenient way to apply a collection of resource settings to all current children of a Spring Box. 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 Spring Box, with equal spacing on either side of the entire group of children.
- **XmSPAN**: Stretches all children equally to fill the entire space of the Spring Box.
- **XmLEFT**: Sets all children to their natural size and moves them to the left edge of the Spring Box. Applies only to **XmNdefaultHorizontalLayout**.
- **XmRIGHT**: Sets all children to their natural size and moves them to the right edge of the Spring Box. Applies only to **XmNdefaultHorizontalLayout**.
- **XmTOP**: Sets all children to their natural size and moves them to the top edge of the Spring Box. Applies only to **XmNdefaultVerticalLayout**.
- **XmBOTTOM**: Sets all children to their natural size and moves them to the bottom edge of the Spring Box. Applies only to **XmNdefaultVerticalLayout**.
- **XmDISTRIBUTE**: Sets all children to their natural size and distributes them evenly across any open space in the Spring Box.
- **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

Spring Box Constraint Resources

Following are constraint resources that are added to children of a Spring Box. 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 Spring Box.

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 Constraint 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 container 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 container.

XmNbottomAttachment, XmNtopAttachment, XmNleftAttachment, XmNrightAttachment

These resources specify the attachment of the bottom, top, left, or right side, respectively, 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.

Paned and HPaned Windows

The Paned Window is a composite container that tiles its children vertically. Children are positioned top-to-bottom in the order in which they are created. The Paned Window grows to match the width of its widest child, and all other children are forced to this width. The height of the Paned Window 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 HPaned Window is a Silicon Graphics extension to Motif that supports horizontal panes. This container is otherwise identical to Paned Window.

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 Paned Window presents an interaction problem when used in a tool such as RapidApp because it stretches its first child to cover the entire window, and you cannot drop additional interface elements directly on the Paned Window itself. There are several solutions to this issue:

Drop on a non-container child or class

If any child of a Paned Window 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 Paned Window. This suggests a work style for creating complex panes: create the collection of interface elements to be placed in each pane separately, define as a class, and add the Paned Window last.

Use Keep Parent Mode

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

Drop on the Sash

Once a Paned Window container has more than one child, you can drop new elements onto a Sash, the small control located between elements to add new panes.

Paned and HPaned Window Resources

The following is the Paned and HPanedWindow resource:

XmNseparatorOn

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

Paned and HPaned Window Constraint Resources

The following are the resources supported by any child of a Paned Window:

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.

Row Column

The Row Column is a general-purpose container capable of containing any widget type as a child. The type of layout enforced by the Row Column 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)

Row Column Resources

Following are the Row Column resources:

XmNadjustLast

If **XmNadjustLast** is set to True, the last row of children is stretched to fill the Row Column to the bottom edge when **XmNorientation** is XmHORIZONTAL. The last column of children is extended to the right edge of Row Column 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 Row Column container. The **XmNentryAlignment** resource controls the type of textual alignment.

XmNnumColumns

Specifies the number of rows or columns supported by the Row Column container. The resource controls the number of elements in the minor dimension; this resource is meaningful only when **XmNpacking** (described below) is set to XmPACK_COLUMN.

XmNorientation

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

- XmNpacking** The value of this resource determines how the row column widget lays out its children. When a Row Column container 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 Row Column container must wrap. The Row Column 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 Row Column container attempts to become large enough to enclose all entries.

Scrolled Window

The Scrolled Window is a container that combines one or two Scroll Bar widgets and a viewing area to implement a visible window onto another (usually larger) data display. The visible part of the window can be scrolled through the larger display by the use of Scroll Bars.

Scrolled Window 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.

Scrolled Window Resources

Following are the resources supported by the Scrolled Window container:

XmNscrollBarDisplayPolicy

Controls the automatic placement of the Scroll Bars. If this resource is set to `XmAS_NEEDED` and if **XmNscrollingPolicy** is set to `XmAUTOMATIC`, Scroll Bars are displayed only if the workspace exceeds the clip area in one or both dimensions. A resource value of `XmSTATIC` causes the Scrolled Window to display the Scroll Bars 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`, Scrolled Window automatically creates the Scroll Bars, attaches callbacks to the Scroll Bars, and automatically moves the work area through the clip window in response to any user interaction with the Scroll Bars.

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

Drawing Area and Visual Drawing

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

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

The Visual Drawing container is a Silicon Graphics extension that differs from the normal Motif Drawing Area in its support for Visual types.

Drawing Area and Visual Drawing Resources

Following are the resources supported by both the Drawing Area and Visual Drawing containers:

XmNexposeCallback

Specifies the member function to be called when Drawing Area 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 Drawing Area window is made smaller.

XmNinputCallback

Specifies the member function to be called when the Drawing Area 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 Drawing Area is resized. The callback reason is `XmCR_RESIZE`.

Visual Drawing Resources

Following are the resources supported by the Visual Drawing container only:

SgNditherBackground

If this resource is `True`, if the visual used with this container is a `TrueColor` or `StaticColor` visual, and if the container is unable to get an exact match for the requested background color, the container attempts to produce a dithered pixmap that produces a closer background to that requested. If one is found, it automatically sets the

XmNbackgroundPixmap resource to this pixmap. See the `SgVisualDrawingArea` reference page for more details.

SgNinstallColormap

If this resource is set to `True`, it specifies that the container should set the `WM_COLORMAP_WINDOWS` property on the shell that contains this widget, so the window manager installs the colormap when the application gets focus. See the `SgVisualDrawingArea` reference page for more details.

Radio Box

The Radio Box is really a Row Column container configured to force one-of-many behavior on its children, which must be toggle buttons. RapidApp creates a Radio Box 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 Radio Box, but then issues warnings at run time. Because the “radio” behavior can be achieved only with toggles, RapidApp supports only toggle children.

Radio Box Resources

All the Radio Box resources are the same as for Row Column, with the following addition:

XmNradioAlwaysOne

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

Frame

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

If you include a title, it is generally best to add the title first. The title is 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 container, and can have the following values:

- XmSHADOW_IN: draws an inset border.
- XmSHADOW_OUT: draws the Frame so that it appears outset.
- XmSHADOW_ETCHED_IN: draws the Frame using a double line giving the effect of a line etched into the window.
- XmSHADOW_ETCHED_OUT: draws the 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. It can have the following values:

- **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.

GLDraw

The GLDraw container creates an empty window suitable for OpenGL drawing. It provides a window with the appropriate visual and colormap needed for OpenGL, based on supplied parameters. GLDraw also provide callbacks for redraw, resize, input, and initialization.

Included in the information provided when creating a GLDraw 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 structure must have been obtained elsewhere; it is the application designer's responsibility to make sure that the structure 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 GLDraw also allocates the colormap unless one is provided by the application. (If a colormap is provided, the application writer is responsible for guaranteeing compatibility between the colormap and the visual). If an application creates multiple GLDraw containers 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 GLDraw 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.

Other resources:

allocateBackground

If `TRUE`, the background pixel and pixmap is allocated if appropriate using the newly calculated colormap and visual. If `FALSE`, they will retain values calculated using the parent's colormap and visual. Applications which wish to have X clear their background for them will usually set this to `TRUE`. Applications clearing their own background will often set this to `FALSE`, although they may set this to `TRUE` if they query the background for their own use. One reason to leave this resource `FALSE` is that if color index mode is in use this avoid using up a pixel from the newly allocated colormap. Also, on hardware that supports only one colormap, the application may need to do more careful color allocation to avoid flashing between the OpenGL colormap and the default X colormap. (Note that because of the way Xt works, the background colors are originally calculated using the default colormap; if this resource is set they can be recalculated correctly. If a colormap was explicitly supplied to the widget rather than being dynamically calculated, these resources are always calculated using that colormap.)

installBackground

If set to `TRUE`, the background is installed on the window. If set to `FALSE`, the window has no background. This resource has no effect unless `GLwNAllocateBackground` is also `TRUE`.

For more information about these attributes and visual types, see the reference pages for the `GLwDrawingArea` widget, the reference page for `glxChooseVisual`, and the OpenGL specification.

Tabbed Deck

The Tabbed Deck container is a composite component that combines a ViewKit **VkDeck** manager and a **VkTabPanel**. You can add items to the Tabbed Deck 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.

Tabbed Deck Resources

forceChildClass

If True, the Tabbed Deck's child is implemented as a C++ class.

Controls Palette

The Controls palette (see Figure A-15) contains control interface elements such as the text field, finder, and scroll bar.

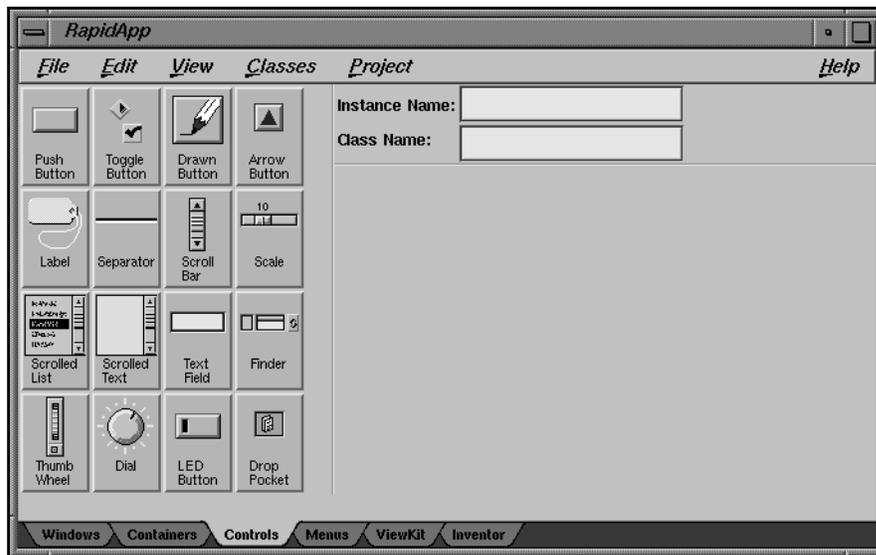


Figure A-15 Controls Palette

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

Push Button

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

Push Button Resources

Following are the resources supported by the Push Button widget:

XmNactivateCallback

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

XmNalignment

Specifies the label alignment for text or pixmap. It can have the following values:

- 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 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. It can have the following values:

- **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 Push Button 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 Push Button:

```
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 Push Button 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 Push Button is a subclass of the Label widget, so the appearance of the Push Button can be manipulated the same as the Label widget. For example, consider the following code:

Example A-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_TAG);
```

Example A-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 A-3 Setting Text on a Subclass of Label Using the VkeZ API

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

Toggle Button

Toggle Button 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 one-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 one-of-many selection state.

Toggle Button Resources

Following are the Toggle Button resources:

XmNalignment

Specifies the label alignment for text or pixmap. It can have the following values:

- **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 the resource is set to True. When the resource is 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. It can have the following values:

- **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 Toggle Button is selected. When the Toggle Button 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 Toggle Button. A value of **False** indicates that the Toggle Button is not set. A value of **True** indicates that the Toggle Button is set. Setting this resource sets the state of the Toggle Button.

XmNshadowThickness

Controls the thickness of the shadow which is useful when simulating radio buttons.

XmNvalueChangedCallback

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

Code Examples

Following are examples of Toggle Button use:

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

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

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

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

Drawn Button

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

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

Drawn Button Resources

Following are the Drawn Button resources:

XmNactivateCallback

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

XmNalignment

Specifies the label alignment for text or pixmap. It can have the following values:

- **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 Drawn Button 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. It can have the following values:

- **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 Push Button.

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.

Arrow Button

The Arrow Button widget is similar to the Push Button widget, but is displayed as a directional arrow.

Resources

Following are Arrow Button resources:

XmNarrowDirection

Determines the arrow direction.

XmNactivateCallback

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

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. It can have the following values:

- **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 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. It can have the following values:

- **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 A-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_TAG);
```

Example A-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 A-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. It can have the following values:

- 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.

Scroll Bar

The Scroll Bar widget allows the user to view data that is too large to be displayed all at once. Scroll Bars are usually located inside a Scrolled Window and adjacent to the widget that contains the data to be viewed. When the user interacts with the Scroll Bar, the data within the other widget scrolls.

A Scroll Bar 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.

Scroll Bar Resources

The following resources are available for the Scroll Bar 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 Scroll Bar 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 Scroll Bar use:

Example A-9 Getting the Value of a Scroll Bar Using the IRIS IM API

```
int value;  
XtVaGetValues(widget, XmNvalue, &value, NULL);
```

Example A-10 Getting the Value of a Scroll Bar Using the VkeZ API

```
int value = EZ(widget);
```

Example A-11 Setting the Value of a Scroll Bar Using the IRIS IM API

```
XtVaSetValues(widget, XmNvalue, 100, NULL);
```

Example A-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 Scroll Bar. 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 A-13 Getting the Value of a Scale Using the IRIS IM API

```
int value;  
XtVaGetValues(widget, XmNvalue, &value, NULL);
```

Example A-14 Getting the Value of a Scale Using the VkeZ API

```
int value = EZ(widget);
```

Example A-15 Setting the Value of a Scale Using the IRIS IM API

```
XtVaSetValues(widget, XmNvalue, 100, NULL);
```

Example A-16 Setting the Value of a Scale Using the VkeZ API

```
EZ(widget) = 100;
```

Scrolled List

Scrolled List 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. Scrolled List operates on an array of compound strings that are defined by the application. Each compound string becomes an item in the Scrolled List, 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 Scrolled Window that contains the List widget. You can select the Scrolled Window by clicking on the Scroll Bar area, or using the “Select Parent” command.

XmNscrollBarDisplayPolicy

Controls the automatic placement of the Scroll Bars. If this resource is set to `XmAS_NEEDED` and if **XmNscrollingPolicy** is set to `XmAUTOMATIC`, Scroll Bars are displayed only if the workspace exceeds the clip area in one or both dimensions. A resource value of `XmSTATIC` causes the Scrolled Window to display the Scroll Bars 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`, Scrolled Window automatically creates the Scroll Bars, attaches callbacks to the Scroll Bars, and automatically moves the work area through the clip window in response to any user interaction with the Scroll Bars.

When **XmNscrollingPolicy** is set to `XmAPPLICATION_DEFINED`, the application is responsible for all aspects of scrolling. The Scroll Bars 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 Scroll Bar is added for a Scrolled List. If this resource is set to XmVARIABLE, the List grows to match the size of the longest item, and no horizontal Scroll Bar 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 Scroll Bar is added for a Scrolled List 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 multiline scrollable text editor.

Scrolled Text Resources

Following are the resources supported by the Scrolled Text 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 this resource is set to True. Prohibits the user from editing the text when this resource is 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 Scroll Bar 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 Scrolled Window 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`.

Text Field

Text Field is a simple, single line text editor. It is similar to the Scrolled Text widget, but can have only a single row of text and is not scrollable.

Text Field Resources

Following are the resources supported by the Text Field 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 this resource is set to True. Prohibits the user from editing the text when this resource is 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 Drop Pocket pocket, a Text Field, 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 Drop Pocket for displaying icons representing entries in the Finder's text field. These icons are Silicon Graphics' environment file icons. File icons from FrameMaker, Searchbook, or similar applications can be dropped on the Drop Pocket.

Finder Resources

Following are the resources supported by the Finder widget:

XmNactivateCallback

This callback is called when a ZoomBar button is pushed, 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**.

Thumb Wheel

Thumb Wheel 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 Thumb Wheel has an elongated rectangular region within which a wheel graphic is displayed. The user can modify the Thumb Wheel's value by spinning the wheel. A Thumb Wheel can also include a *Home* button located outside the wheel region. This button allows the user to set the Thumb Wheel's value to a known position.

Thumb Wheel Resources

Following are the resources supported by the Thumb Wheel 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 Thumb Wheel vertically or horizontally. This resource can have values of XmVERTICAL and XmHORIZONTAL.

XmNunitsPerRotation

Specifies the number of units in a rotation.

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:

SgNangleRange

Determines how far the valuator can move.

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.

XmNhighlightThickness

Controls the thickness of the highlight area around the dial.

SgNmarkerLength

Specifies the size of the dial’s markers. To hide the markers, set this to zero.

SgNangleRange

Specifies the angular range, in degrees, through which the dial is allowed to rotate. This, with **XmNmaximum** and **XmNminimum**, controls the fineness or coarseness of the dial control.

SgNdialVisual Specifies the look of the dial, either SgKNOB or SgPOINTER.

XmNshadowThickness

Controls the thickness of the shadow.

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`.

LED Button

LED Button is similar to a toggle button in that it is a two-state button. However, instead of a square or diamond-shaped indicator to the left of text or a pixmap, the LED Button is a pushbutton-style widget with a built-in indicator “light.” When the LED Button is set, the indicator is bright; when it is unset, the indicator is dim.

LED Button Resources

Following are the LED Button 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 indicator light 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 LED Button is selected. When the LED Button 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 LED Button. A value of False indicates that the LED Button is not set. A value of True indicates that the LED Button is set. Setting this resource sets the state of the LED Button.

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

Drop Pocket

The Drop Pocket widget is designed to receive desktop icons from the IRIS Indigo Magic desktop. The Drop Pocket displays the file icon as a visual reminder of the file associated with the Drop Pocket. See the SgDropPocket reference page for more details.

Drop Pocket Resources

Following are the resources for Drop Pocket:

SgNiconUpdateCallback The member function to be invoked when an icon is dropped in the Drop Pocket. 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 Drop Pocket.

Menus Palette

The Menus palette (see Figure A-16) contains menu interface elements such as pulldown menu, option menu, and menu separator.

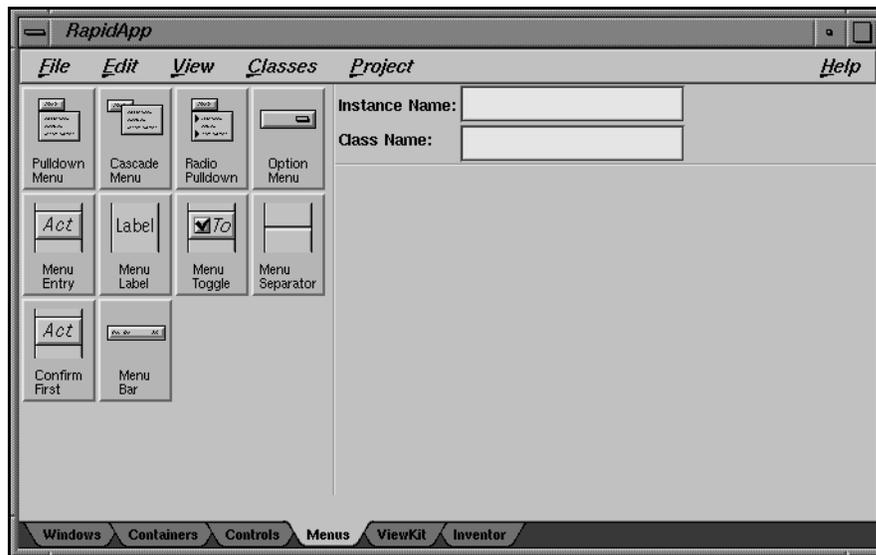


Figure A-16 Menus Palette

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 the menu is displayed, you can add items (MenuEntry, MenuLabel, MenuToggle, MenuSeparator, ConfirmFirst, or other pulldowns) by dropping new elements on the displayed menu area. You can dismiss the menu by clicking the pulldown (not the displayed menu pane).

Pulldown Resources

Resources in the Pulldown menu correspond to the visible menu entry for this pulldown. The following resources are available:

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 the menu is displayed, you can add items (MenuEntry, MenuLabel, MenuToggle, MenuSeparator, ConfirmFirst, or other pulldowns) by dropping new elements on the displayed menu area. You can dismiss the menu by clicking the pulldown (not the displayed menu pane).

Note: For experienced IRIS IM developers: This item is identical to the Pulldown menu item, and is present as an aid to those less familiar with the IRIS IM menu structure.

Cascade Resources

Resources in the Cascade menu correspond to the visible menu entry for this menu pane. The following resources are available:

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 added 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 the menu is displayed, you can add items (MenuEntry, MenuLabel, MenuToggle, MenuSeparator, ConfirmFirst, or other pulldowns) by dropping new elements on the displayed menu area. You can dismiss the menu by clicking the pulldown (not the displayed menu pane).

RadioPulldown Resources

Resources in the RadioPulldown menu correspond to the visible menu entry for this menu pane. The following resources are available:

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 used 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 the menu is displayed, you can add items (MenuEntry elements) by dropping new elements on the displayed menu area. You can dismiss the option menu by clicking on the menu button (not the displayed menu pane).

OptionMenu Resources

The following OptionMenu resource is available:

XmNLabelString

Determines the value of an optional string to be displayed to the left of the option menu as a title. If this resource is 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 nonselectable entry. The `MenuLabel` is represented in the program as a ViewKit **`VkMenuLabel`** object.

MenuLabel Resources

The following `MenuLabel` resource is available:

`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 is represented in the program as a ViewKit **VkMenuSeparator** object.

Confirm First

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 ConfirmFirst is 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.

Menu Bar

The Menu Bar item creates a menu bar for use in a custom dialog. You can add this menu bar to dialogs only. If you want a menu bar in a main window, create a `VkWindow`.

This menu bar doesn't contain the standard menu bar entries provided with a `VkWindow` menu bar; instead, it contains only two dummy menu panes with three dummy menu entries each. You can display a menu pane by selecting it, and then clicking it once again. Once displayed, you can add additional items (`MenuEntry` elements) by dropping new elements on the displayed menu area. You can dismiss the menu pane by clicking the menu button (not the displayed menu pane) again.

ViewKit Palette

The ViewKit palette (see Figure A-17) contains ViewKit interface elements such as tab panel, tick marks, and graph.

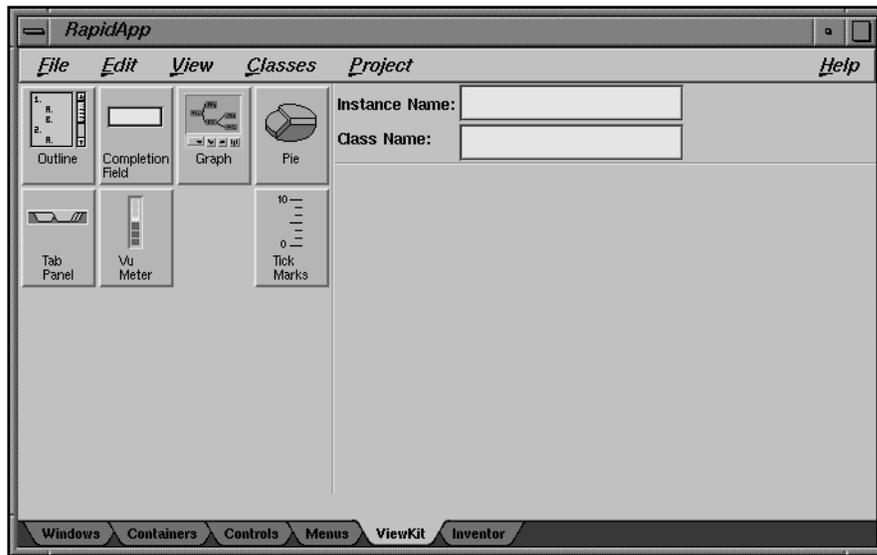


Figure A-17 ViewKit Palette

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 be done only 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.

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.

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 a horizontal orientation.

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.

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 A-18) contains Inventor interface elements such as material list, light slider set, and render area.

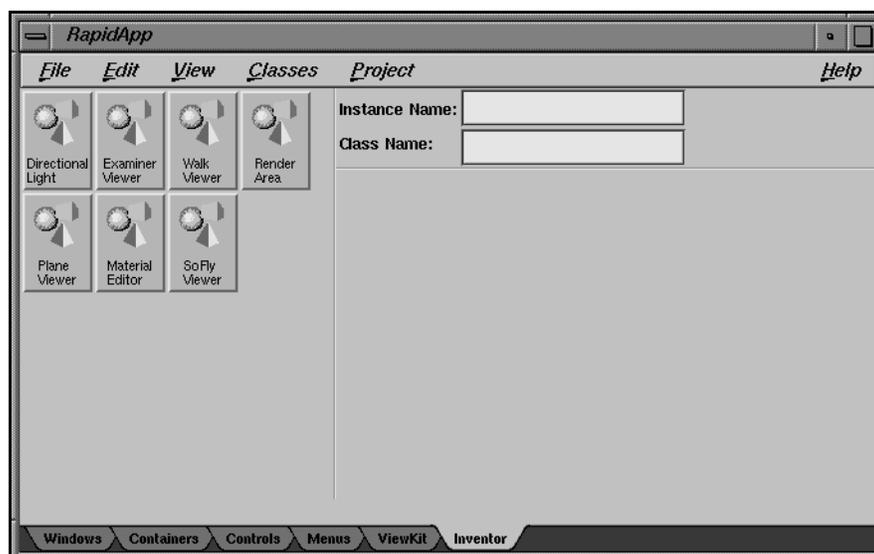


Figure A-18 Inventor Palette

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

Directional Light

This Inventor class is used to edit a **SoDirectionalLight** node (color, intensity, and direction are changed). In addition to directly editing directional light nodes, the editor can also be used with callbacks that are 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 `SoDirectionalLight(3)` reference page or the *Inventor Mentor* for more details.

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 farther away from the point of interest. The viewer also supports seek to quickly move the camera to a desired object or point. See the `SoXtExaminerViewer(3)` 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 viewer.

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 viewer on 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(3)` 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 appears only while in viewing mode (default is off).

feedbackSize

Set the point of rotation feedback size in pixels (default 20 pixels).

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—similar to an elevator. See the `SoXtWalkViewer(3)` 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 viewer on 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 <code>SoXtViewer(3)</code> 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.

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 viewer on 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(3)** 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** Sets whether the viewer is turned on or off. When it is turned on, events are consumed by the viewer. When viewing is off, events are processed by the viewer’s render area. This means events are sent down to the scene graph for processing (in other words, picking can occur).

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 forward/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 that 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 axis. See the `SoXtPlaneViewer(3)` 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 viewer on 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 <code>SoXtViewer(3)</code> 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 a **SoMaterial** node. The editor can also be used directly with 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 `SoXtMaterialEditor(3)` 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 `SoXtFlyViewer(3)` reference 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 viewer on 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(3)` 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** Sets whether the viewer is turned on or off. When it is turned on, events are consumed by the viewer. When viewing is off, events are processed by the viewer's render area. This means events are sent down to the scene graph for processing (in other words, picking can occur).

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 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.

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 such as *core*, **.o*, and so on. You can add to this list by 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 definitions 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 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

```
OPTIMIZER=-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

Frequently Asked Questions and Tips

This appendix contains the following sections:

- “Frequently Asked Questions,” answers the most common questions about using RapidApp
- “RapidApp Tips,” provides tips for using RapidApp

Frequently Asked Questions

This section answers frequently asked questions (FAQs) about RapidApp’s operation. The questions are divided into the following alphabetical categories:

- “Bitmap/Pixmap Icons”
- “Code”
 - “Client Data”
 - “Leading Underscores”
 - “Non-C++ Languages”
 - “Portability”
 - “User-Edited Code”
- “Containers”
 - “Size”
- “Dialogs”
 - “And Buttons”
 - “Creation”
- “Editors”

- “Help”
 - “Help System”
 - “Hiding Help”
- “Interface Elements”
 - “Access”
 - “Alignment”
 - “Appearance”
 - “Creation”
 - “Names”
 - “Resources”
 - “Selection”
 - “Size”
 - “User-Defined Widgets”
- “Inventor”
 - “Delayed Creation”
- “Menus”
 - “Grayed-Out Menu Items”
 - “Manipulation”
- “RapidApp Files”
 - “Adding Company Standard Header”
 - “Makefile”
- “TCL Support”
- “Tips”
 - “Accessing”
 - “Removing”

- “User-Defined Classes”
 - “Class Connection”
 - “Dynamic Behavior”
 - “Editing”
 - “Functionality”
 - “Import”
 - “Instantiation”
- “ViewKit”
 - “Accessing”
 - “Delayed Creation”
- “VkEZ and EZ Functions”
- “Windows”
 - “And Elements”
 - “Resizable Layouts”

Bitmap/Pixmap Icons

- How do I create icons for bitmaps or pixmaps?

Any number of external tools work well for icon creation. For example:

 - The “bitmap” program distributed with *x_eoe* is a reasonable bitmap editor.
 - Any drawing editor can be used to create color icons. If necessary, convert the icons using the *ppm* or similar utility.
 - The *xpaint* program, available as part of the original IndiZone package, is good for editing pixmaps.
 - The public domain *xv* editor is useful for reading and processing various file types and writing the Xpm format recognized by IRIS IM (and RapidApp).

Code

Client Data

- How do I provide client data to my callbacks?

Client data is mostly meaningless in the style of code generated by RapidApp. In C, it is necessary to pass state around to callbacks via client data. 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 client data would be contained within another class, and therefore passing it in some way would be a violation of encapsulation. The only reasonable use of client data would be to pass a simple value, such as an integer code or string to provide information about the context of the call. While occasionally this is useful, there are other ways to deal with this need (different callbacks that call a second function, for example).

Leading Underscores

- Why does RapidApp-generated code use leading underscores for data member names?

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 partly 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 a convention used throughout ViewKit and happens to lend itself well to code generation.

Non-C++ Languages

- I want to use C (Ada, Fortran, Pascal, Cobol, ASM, TCL, etc.). 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.

Portability

- How portable is the code generated by RapidApp?

This depends on what elements you use. If you use only the standard IRIS IM widgets, the dependencies are X, Xt, IRIS IM, and ViewKit. As long as you have these libraries on your target platform, it should be possible to recompile the code generated by RapidApp. The desktop icons, FTR files, and so on, are specific to Silicon Graphics.

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

Several widgets are unique to Silicon Graphics, 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.

User-Edited Code

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

Not if you understand RapidApp's strategy for merging code. RapidApp uses a comprehensive strategy for merging code that it generates with existing code. 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 problem areas by hand. You can help the merge process proceed smoothly by limiting your changes to the provided "user code block" areas, not making gratuitous changes (like reformatting for style) to the generated code, placing large bodies of code in external files, and so on. See "Code Merging" on page 58 for more information.

Containers

Size

- Why do some containers change size when I add or remove children?

Each container has its own algorithm for arranging its children. This algorithm is triggered each time a child is added or removed. Even though you specify a size for a container, the container itself may recompute and change this size when its contents change. This is central to the behavior of IRIS IM and there is little that can be done to change it. For more information, see “Moving and Resizing Interface Elements” on page 29.

Dialogs

And Buttons

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

In RapidApp, you cannot. The **VkDialogManager** class and all subclasses allow you to determine what buttons appear when the dialog is posted. See Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide*.

Creation

- How do I create a standard IRIS IM dialog RapidApp?

Dialogs tend to be dynamic by nature. Although RapidApp could allow you to associate an 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 a condition that can be determined only at runtime. Therefore, RapidApp doesn’t provide a way to create 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, simply 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 on dialogs, see “Work With Dialogs” on page 97 in this guide and Chapter 7, “Using Dialogs in ViewKit,” in the *IRIS ViewKit Programmer’s Guide*.

- How do I create a custom dialog?

You create your own dialog by selecting one of the dialog windows from the Windows palette. Add a single container and design the contents of your dialog. The resulting class is derived from an IRIS ViewKit dialog class and supports the same API as other ViewKit dialogs. See “Work With Dialogs” on page 97 for more information.

Editors

- How do I launch my own editor from RapidApp?
 1. Set the environment variable \$WINEDITOR
 2. From the File menu of RapidApp’s main window, choose “Preferences.”
 3. In the RapidApp card, set the “Use \$WINEDITOR” option.

Your window-based editor is launched when you choose “Edit File...” from the Project menu.

Help

Help System

- How do I hook up a Help system to a program RapidApp creates?

Your program makes 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 best option is to use the InSight-based Silicon Graphics help system. See Chapter 9, “Providing Online Help With SGIHelp,” in the *Indigo Magic Desktop Integration Guide* for information on how the Silicon Graphics online help system works, how it interprets the help token it receives, and how to provide online help for your application.

The *vkhelp* library distributed with ViewKit provides 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. See Appendix C, “Using a Help System with ViewKit,” in the *IRIS ViewKit Programmer’s Guide* for information on the ViewKit links to a help system and the *vkhelp* library.

Hiding Help

- How do I disable the Help menu for my application, which doesn’t include any online help?

Select the window in question and set the `hideHelpMenu` resource to `True`.

Interface Elements

Access

- How do I programmatically access the widgets encapsulated in a C++ class created by RapidApp?

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. 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 can write an access function for the label element and use `XtSetValues()`, and so on, 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 is 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 can 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.

Alignment

- How do I align two or more interface elements?

The best way to align interface elements in RapidApp is to select the appropriate layout containers. The RowColumn container is often used for this. Also consider the Spring Box container, which provides easy and powerful control over alignment of its children. For more information on choosing the right container see “Work With Containers” on page 72.

You can also use the universal grid settings found in the View menu to help you with alignment.

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

Alignment tools make sense only when applied to multiple interface elements, and in RapidApp you cannot select multiple elements. RapidApp does offer a grid, found in the View menu, to help with alignment.

Appearance

- Why don't some of my elements, such as labels, appear when I run programs created with RapidApp?

X applications use resource files; files containing widget resources. When an application is running, it uses these resource files to configure its widgets appropriately. RapidApp creates a resource file for your application automatically and stores it in the designated project directory. Unfortunately, X doesn't look in the current directory when searching for an application's resource file. You need to direct the search path using one of the following methods:

- To add the current directory to the application's search path, set the environment variable XUSERFILESEARCHPATH to “%N%S”. You might want to do this as part of your login setup.
- To guarantee that the resource files are found while testing your application (if you haven't set the environment variable as instructed above), run the application from RapidApp's Project menu.

- Install the *app-defaults* file in */usr/lib/X11/app-defaults*. If you enter `make install` in your application's project directory, the *Makefile* generated by RapidApp does this for you automatically. If you choose this approach, remember to reinstall any time you make changes.
- To cause the resources to be placed in each class as defaults, set the option "Place resources in classes" to true. This option is found on the Code Style card in the Preferences dialog accessed from the File menu.

Creation

- Sometimes when I drop a widget on a location, the widget is created in a new window instead of where I dropped it. 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.

Names

- Sometimes I enter a name in RapidApp and RapidApp adds a number after it (for example, "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. Rather than causing an error later, RapidApp enforces this convention when you enter the names. For example, if you have one button with an Instance name of "OK" and you give that same name to another button, RapidApp changes it to the unique name of "OK1". And, if you provide a callback function called **OK()**, RapidApp again makes the function name unique and you end up with **OK2()**. You can avoid this behavior by adopting naming conventions for your interface elements. For example, the *OK* button, might have the label "OK," the name "okButton," and a callback function **ok()**. (The label doesn't need to be unique.)

Resources

- I want to apply a resource to multiple interface elements. Is there an easy way to do this?

There is no way to select multiple interface elements, or apply a resource to a group of existing elements. However, if you plan ahead you can create an element, set its attributes, and then copy and paste the element. The copied element retains the attributes of the original.

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

You can determine a widget's full set of resources by looking at the reference pages for the IRIS IM widget or by consulting an IRIS IM reference guide. RapidApp displays the most commonly used resources. You can set any resource in your program or in the *app-defaults* file.

- How do I set colors for my interface elements?

In the Indigo Magic Desktop environment, colors are controlled by schemes. You can override these schemes either programmatically or in your *app-defaults* file. By default, RapidApp creates programs whose colors are determined solely by schemes. You can always programmatically set colors for interface elements. All elements are available to derived classes as protected data members, so it's easy to override colors. You can also override the scheme settings by setting colors in your *app-defaults* file. Simply specify the class name of your program as part of the resource.

If you don't like the scheme being applied, change the scheme your programs use with the scheme browser available from the Toolchest. For information on 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*.

- How do I change the font used by my interface elements?

In the Indigo Magic Desktop environment, fonts are controlled by schemes. You can override these schemes either programmatically or in your *app-defaults* file. However, you are encouraged to use the scheme-specified fonts. In cases where it makes sense, RapidApp allows you to switch the font of individual elements, such as labels, text, and lists, to one of the scheme-supported fonts. For example, you might want to change a label element from bold to normal, or a text element to a fixed-width font.

Selection

- Can I select multiple interface elements?
No, RapidApp doesn't currently support this.

Size

- Why can't I resize an option menu widget?
The IRIS IM Option menu is really a RowColumn widget with a menu floating inside it. You can resize the outer RowColumn widget, but the inner, visible option menu is totally controlled by the RowColumn. This visible option menu is not accessible to RapidApp and cannot be resized.
- Why can't I resize a scale widget?
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 controlled by the Scale. This visible Scrollbar can be horizontally resized or fully resized by modifying the **scaleWidth** and **scaleHeight** resources.

User-Defined Widgets

- Can I add my own widgets to RapidApp?
No, RapidApp does not support the addition of custom widgets at this time. However, you can add ViewKit components created in RapidApp.

Inventor

Delayed Creation

- Why does it take so long to create some entries from the Inventor palette?
Some of the elements on these and other palettes are dynamically loaded from shared libraries as needed. The first time one of these libraries is loaded, the symbols in that library must be resolved before proceeding. This operation can cause a slight delay.

Menus

Grayed-Out Menu Items

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

These menu items invoke other Developer Magic tools and are available only if you have the tools installed. For example, you can't build, debug, or run if you don't have the Build Analyzer (*cvbuild*) installed. The Browse Source menu item invokes the Static Analyzer (*cvstatic*), which must be installed. In addition, for Browse Source to work, you must set the "Generate CVstatic Database" option found on the RapidApp card in the Preferences dialog. When active, the *Makefile* automatically generates a static analysis database. While this is useful, this greatly slows down the compilation, so the option is off by default.

Also, If RapidApp doesn't have a license, items are grayed out.

Manipulation

- How do I add a menu bar to my window?
 - If you want a window with a menu bar, create a `VkWindow` element found on the Windows palette. You can add, remove, or alter the built-in menus as you wish. For more information, see "VkWindow" on page 180.
 - If you want a custom dialog with a menu bar, create a dialog from the Windows palette and add a Menu Bar element from the Menus palette. For more information, see "Creating a Menu Bar" on page 93.
- How do I add items to a menu?
 1. Access the menu pane by clicking the menu entry on the menu bar, or the option menu, depending on the type of menu you are working with.
 2. Click again to display the menu pane.
 3. Drop new elements onto the pane.

The mouse pointer is the hotspot when dropping items.

Dismiss the menu pane by clicking on the menu entry. For more information, see "Work With Menus" on page 92.

- How do I move menu elements within a menu pane?
 - Select the element and choose “Up/Left” or “Down/Right” from the Edit menu of RapidApp’s main window.
 - or–
 - Use the arrow keys to reorder elements in a menu.
- How do I move menu elements between menu panes?

To move the element:

 1. Select the element and choose “Cut” from the Edit menu of RapidApp’s main window.
 2. Choose “Paste” from the Edit menu and click inside the new menu pane.

–or–

 1. Select the element.
 2. Drag and drop the element between menu panes using the middle mouse button while holding down the <ctr1> key.

To copy the element:

 1. Select the element and choose “Copy” from the Edit menu of RapidApp’s main window.
 2. Choose “Paste” from the Edit menu and click inside the new menu pane.

–or–

 1. Select the element.
 2. Drag and drop the element between menu panes using the middle mouse button while holding down the <ctr1> key.

RapidApp Files

Adding Company Standard Header

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

Yes, provide the name of a file to be inserted in the “Header Comments” field in the Project card of the Preferences dialog. The file should be properly formatted, with C++ comment characters.

Makefile

- 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, add externally-created files to the USERFILES list. List only the source files; the *Makefile* does the rest.

TCL Support

- 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.

Tips

Accessing

- How can I browse the tips that RapidApp displays at startup?

See “RapidApp Tips” on page 271.

Removing

- How can I get rid of the tips that show at startup?
 1. From the File menu, choose “Preferences.”
 2. In the RapidApp card, unset the “Show tips on startup” option.

User-Defined Classes

Class Connection

- 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 (UIKit-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 UIKit support for callbacks.

Say you have two classes, **Input** and **Output**, and **Input** has a text field. 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 code the connection, using your favorite editor:

Add a member function **newText()** to class **Output**. This member function does 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()**.

- Use UIKit callbacks:

Add a private or protected member function to **Output**:

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

Add the following line at an appropriate place in the **Output** class where you want to set up the callback:

```
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() is invoked.

Dynamic Behavior

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

The best way is also the simplest: use your favorite editor to add data members, member functions, and so on. Keep your changes within the defined user code blocks. The merge feature of RapidApp assures that your changes are maintained.

If you want to add resources to components that you load onto a RapidApp custom palette, see “Adding Resources to Components” on page 136 for instructions.

Editing

- How can I change a class after I have created it?
 1. Switch to class edit mode by choosing “Edit Classes...” from the Classes menu.
 2. From the displayed list, select a class
This displays the elements of the class.
 3. Edit or manipulate each element as necessary.
 4. Exit class edit mode by clicking the *Leave Class Mode* button.
Changes apply to all instances of that class.

Functionality

- In RapidApp, how do I create a C++ class with functionality and have that functionality saved with the class?

See Chapter 8, “Component Libraries,” for instructions on creating libraries of components that you can reuse and distribute to other developers.

Import

- How do 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:

1. Define your classes.
2. Delete all instances.
3. Save the file.

Only the class descriptions are saved.

4. The next time you use RapidApp, and after you've begun creating your application, load the your classes by choosing "Import" from the File menu.

Instantiation

- Once I've defined a class, how do I add an instance of the class to an existing container?

When you create a class, an icon representing that class is added to the User-Defined. You create instances of the class by clicking on the icon and clicking in any container (just as you do with any other palette icons).

Also, your original collection of interface elements becomes an instance of the new class. You can move this instance into another container by dragging it to the container while holding down the middle mouse button.

ViewKit

Accessing

- 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 more about ViewKit? Is there documentation?

The *IRIS ViewKit Programmer's Guide* is available online if you install the **.books.** subsystem of IRIS ViewKit. You can also order a hard copy of the manual. Reference pages are also part of the ViewKit distribution.

Delayed Creation

- Why does it take so long to create some entries from the ViewKit palette?

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

VkEZ and EZ Functions

- What is VkEZ and why would I use it?

VkEZ is a simple set of wrappers that can be attached to a widget at runtime to provide an easy-to-remember API for manipulating widgets. VkEZ has many of the benefits of “widget wrappers,” in that it offers a C++-like API for manipulating widgets, without 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);
```

See Chapter 7, “VkEZ Library,” for more information.

Windows

And Elements

- Why can't I add elements other than containers to a window element?

Each element on the Windows palette can contain exactly one child which must be a container element or a complex component. If you want to add additional elements, you need to add a container to the window and then add your elements to the container. For information on choosing the correct container, see "Work With Containers" on page 72.

Resizable Layouts

- How do I create resizable layouts?

The container you choose determines the layout style of your window. For layouts that resize, consider the following containers:

- Form: the most frequently used container for resizable layouts. You specify attachments to determine how each child resizes. For information, see "Form" on page 82. For detailed information, consult a Motif book.
- Rubber Board: easy to use, but has some limitations. For more information, see "Rubber Board" on page 74.
- Spring Box: offers a resizable layout that is the entire basis of some toolkits. For more information, see "Spring Box" on page 79.
- Paned Window: offers a limited form of resizable layout. For more information, see "Paned Windows" on page 86.
- RowColumn: offers limited resizability when its **adjustLast** resource is set to true. For more information, see "RowColumn" on page 88.

Generally, applications use a combination of all these widgets. See "Work With Containers" on page 72 for information on selecting and using the various containers provided in RapidApp.

RapidApp Tips

This section includes the following:

- “Displaying Tips”
- “List of Tips”

Displaying Tips

By default, RapidApp’s startup screen contains a “tip,” a suggestion for how to use RapidApp. You can control whether or not the startup screen displays tips. To do so:

1. In the RapidApp main window, from the File menu, choose “Preferences...”
2. In the Preferences dialog, go to the RapidApp card.
3. Unset the “Show tips on startup” option.

The next time you launch RapidApp the startup screen will not include a tip.

List of Tips

The following is a complete list of tips:

- Holding down the Control key allows you to operate on the currently selected widget without accidentally changing the selection.
- You can select a selected widget’s parent by choosing “Select Parent” from the Edit menu, by typing `<Ctrl-P>`, or by holding both `<Shift-Ctrl>` while clicking.
- You can edit the contents of Pulldown and Option menus by selecting the menu and choosing “Show Menu” from the Edit menu or by clicking on the menu while it is selected.
- Many resources are stored in the *app-defaults* file. This file must be installed in an appropriate location before the resources can take effect.
- Generating code does not automatically save the interface you have created. Be sure to choose “Save” or “Save As...” from the File menu before quitting.
- Modified text fields have a different background color until you press `<Return>` or until the text field loses focus.

- The Windows palette contains various top-level windows, the Containers palette contains all manager widgets and the Controls palette contains all control widgets.
- To create a collection of one-of-many toggle buttons, choose a RadioBox container from the Containers palette and add Toggle Button widgets from the Controls palette. Two buttons are provided by default to get you started. You can delete them, edit them, or add to them.
- The RubberBoard widget is an easy-to-use container for creating stretchable layouts. Start with a small window and position all children. Select the RubberBoard and set the setInitial resource to True. Resize the RubberBoard to a large size, reposition all children, and set the setFinal resource to True. The children are resized/repositioned using interpolation.
- Labels, accelerators, and mnemonics are automatically placed in the application's *app-defaults* file.
- Interfaces created with RapidApp automatically use the Indigo Magic look and feel, including using schemes for all colors and fonts. See the schemes documentation in the *Indigo Magic Integration Guide* for information on manipulating schemes and using colors in your application.

Note: You can always programmatically change fonts and colors in your application as well as set them in your *app-defaults* file. But schemes provide a good starting point. You can alter the font used by some elements by selecting a different schemeFont resource.

- Some knowledge of Motif makes it much easier to use RapidApp. It is particularly useful to understand how Motif's container widgets and geometry management work. Refer to a Motif tutorial if you have trouble using the various container widgets.
- If you accidentally resize a widget to the point that it is too small to easily manipulate, you can increase its size by choosing "Grow" from the Edit menu or by typing `<Ctrl-G>`.
- When using a Form container, you can pop up a menu of possible attachments by pressing the right mouse button over one of the attachment icons shown in the selected widget.
- When using a Form container, you can attach a widget to another widget by pressing the left mouse button over an attachment icon and dragging the rubberband line to the intended attachment point.

- When using a Form container, you can specify “attachSelf” by pressing the left mouse button over an attachment icon, dragging the rubberband line away from the widget, then dragging the line back onto the widget itself.
- When using a Form container, you can adjust the offset by moving/resizing a widget, or by holding down `<shift>` while pressing the left mouse button and dragging. A popup menu shows the offset in pixels.
- It is often best to create a complex interface by constructing smaller sections using a single container widget, then placing the completed subsections into the larger layout.
- The Project card in the RapidApp Preferences dialog allows you to specify the name and class name of your application, as well as other useful options. Access this dialog from the File menu.
- You can often change an existing widget to another type. Replace the value of the “Class Name” field in the instance header with the desired widget class and press `<Return>`.
- The Bulletin Board widget is the easiest container to use if you don’t care about resizable layouts and/or internationalization.
- The Rubber Board container widget is the easiest way to create resizable layouts, if you don’t care about internationalization.
- You can rearrange widgets within a RowColumn container or menu pane by choosing “Up/Left” and “Down/Right” from the Edit menu. You can also use the arrow keys.
- You can view the widget hierarchy of a panel built inside RapidApp using the *editres* utility.
- For fast building, browsing and debugging, the Project menu provides an interface to the rest of the Developer Magic environment tools.
- If you would like your application to exhibit the “runonce” behavior seen on many of the Indigo Magic desktop control panels, check “Use RunOnce” on the Project card in the RapidApp Preferences dialog accessed from the File menu.
- You may need to resize a Frame widget slightly to force its children to position correctly.
- Saving often prevents loss of data in case of an unexpected exit from any cause.
- The RubberBoard, SpringBox, Dial, ThumbWheel, Finder, and GlxMDraw widgets are not part of standard Motif, but are SGI extensions to Motif.

- Only the most commonly used widget resources can be modified using RapidApp's resource editor. However, all Motif resources can still be set programmatically or in resource files.
- The Delete operation deletes a widget or widget hierarchy, but does not place it on the clipboard.
- Each Motif container enforces a specific layout algorithm. Choosing the right container makes your task easier. For example, the RowColumn container arranges widgets in rows and columns, and does not allow arbitrary movement of the widgets.
- Changing the name of a widget changes both its resource name and the name of the variable used to represent the widget in any generated code.
- Changing the label of a button or label widget does not affect the name of the widget in the program, only the label displayed by that widget.
- RapidApp requires all names to be unique, and automatically appends numbers to any names already used: label1, label2, label3, and so on. Avoid this by choosing unique names.
- It is easiest to create a labeled frame container by creating the frame, then adding the label. Construct the contents of the frame separately and add it last.
- Selecting a palette icon with the left mouse button results in a rubber-banding outline that allows you to position and resize the widget by holding down the left mouse button and sweeping out the widget's size before releasing the mouse button.
- Motif widgets were not designed for use with an interactive builder and often exhibit odd behavior when manipulated interactively. It is sometimes helpful to resize a window to trigger Motif's layout routines explicitly.
- 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 behave as expected.
- It is often useful to construct a complex interface bottom-up, by creating small simple interfaces that can be combined to form larger more complex interfaces.
- When you generate code, the *app-defaults* file sets the `sgiMode` resource to True so your application automatically runs with the SGI look and feel.

- You can “show” more than one menu pane at a time, and even drag items between menu panes. Click on a menu and choose “Show Menu” from the Edit menu on RapidApp’s main window, or simply click on the menu again — either shows the menu pane. You can dismiss the menu pane by choosing “Show Menu” again, or by clicking on the displayed menu.
- When you test a program by choosing “Run Application...” from the Project menu, RapidApp ensures that the *app-defaults* file is correctly loaded. If you run the program from a shell, be sure to set `XUSERFILESEARCHPATH` to `%N%S` or install the resource file in `/usr/lib/X11/app-defaults`. You can also copy or link the *app-defaults* file to your home directory while testing.
- For best results, modify code only in the areas marked as being outside the generated code sections.
- RapidApp treats all widget collections as classes (components). To avoid generated names, explicitly declare the classes you want to use.
- Your application can automatically be ToolTalk-smart. On the Project card in the RapidApp Preferences dialog, set the Message system to ToolTalk. Access this dialog from the File menu.
- You can make changes to any generated file. RapidApp merges all changes each time code is generated.
- You can produce inst images for your completed application by simply typing **make image**. The images are generated in an images subdirectory. Choose “Edit Installation” from the Project menu to make changes to the installation.
- The `VkTabbedDeck` allows you to stack components on top of each other so they can be selected one at a time, like RapidApp’s Palette area. It’s best to create the components before placing them in the Deck.
- You can specify a pixmap or bitmap for labels and buttons. Enter the name of an Xpm pixmap or bitmap file in the `labelPixmap` field. The `XmNlabelType` switches to `XmPIXMAP` automatically. The pixmap or bitmap is placed in the *pixmap.h* file and directly included in your program.
- You can remove a component from a palette by choosing “Unmake Class” from the Classes menu and responding to the dialog that appears. You can also choose “Edit Classes” from the Classes menu, display the class, and delete it.
- You can save collections of components to be loaded into other projects. Just create one or more components, remove all instances of the component(s), and save to a file.

- The `VkWindow` class creates a ready-made menubar with the most common menu items specified by the SGI Style Guide. You can add to or delete entries from these menus.
- You can have a `cvstatic` database automatically created for you when you compile. Just check “Generate `Cvstatic` Database” in the RapidApp card of the Preferences dialog.
- Classes should normally consist of a widget hierarchy: A container and its children.
- If you set the environment variable `XUSERFILESEARCHPATH` to `%N%S`, applications run correctly from their development directory without the need to install the application resource file.
- To display an image, specify a bitmap or pixmap as the `labelPixmap` resource for any label or button widget. You can specify a full or relative path.
- It is best to start each project in a clean directory. RapidApp assumes a one-project/one-directory model.
- If you change the name of a class within RapidApp after you have made changes to the source code, you need to merge your changes into the new class by hand.
- If you move a project directory, you should be sure to also move all hidden files and directories. RapidApp creates various files and directories whose names start with a “.” that are needed to maintain your code. The hidden directories are also important if you use configuration management, or share a project between multiple users.
- If you wish to remove all files in a project, remember to remove the hidden files and directories (those whose names begin with a “.”) as well. If you start a new project in a previously-used directory, RapidApp could become confused by the presence of the old files.
- RapidApp automatically creates the files needed to create `inst` images. You can execute “make image” to create installable images. “Edit Installation” in the Project menu allows you to edit the information used to build the `inst` images.
- RapidApp automatically creates the files needed to integrate with the Indigo Magic desktop. These include an `ft` file that contains file typing rules, a sample icon file, and commands in the `inst` files. You can edit your icon using `IconSmith`. If you plan to distribute your application to others you should obtain your own desktop tag. To get your own unique tag, send mail to `desktoptags@sgi.com`.

- You can add dynamic behavior to your application using the Developer Magic Debugger's Fix+Continue feature. Choose "Debug Application..." from the Project menu, run your program from the debugger, and then exercise the portion of the interface you wish to complete. If you have a callback associated with the user action, the debugger automatically stops in `::VkUnimplemented()`. Return from this function and then use the commands in the Fix+Continue menu to add and execute code for your program.
- RapidApp works best if you avoid making gratuitous changes to the generated code. For best results, avoid modifying the code outside designated areas.
- RapidApp includes a set of convenience routines in a VkeZ library. This library allows you to connect various widgets without knowing as much as the Motif API requires. For example, to display the value of a Scale widget in a Label widget, you can write:

```
EZ(_label) = (int)EZ(_scale);
```

To append the text in a label to the value of a scrollbar and display the results in a TextField widget, you can write:

```
EZ(_text) << EZ(_label) << EZ(_scrollbar);
```

The VkeZ library included with RapidApp is useful for prototyping, but is very inefficient. You should plan to replace any uses of the EZ interface with a more direct approach as soon as possible.

- To activate or deactivate the sounds used by RapidApp, toggle the "Enable sound" option on the RapidApp card in the RapidApp Preferences dialog, accessed from the File menu.
- To configure various application-specific parameters, use the RapidApp Preferences dialog, accessed from the File menu. Some of these options apply only to the current program. Others, such as your preferred file suffixes apply to all future uses of RapidApp, unless changed.
- When you add your own files to the *Makefile* generated by RapidApp, you should add them to the USERFILES list. This makes it easier for RapidApp to merge any changes to the *Makefile* during development.
- Clicking on a pulldown menu while it's selected, displays its contents for editing. Also, this technique works with cascading and option menus.
- You can "clone" an existing interface by dragging and dropping with the middle mouse button while holding down `<ctr1>`. Cloned objects retain all the characteristics (resources) of the original element.

- Option menus are a composite of many different widgets. The base of the option menu is a RowColumn widget, which occupies the area around the visible button. Attempts to resize an option menu appear to be unsuccessful because you can only resize the base RowColumn widget, not the visible button.
- You can edit the contents of an option menu by clicking on the button area after selecting the option menu. You can also select the option menu and choose “Show Menu” from the Edit menu.
- You can add a title to the left side of an option menu by setting the option menu’s labelString resource. Do not confuse this resource with the labelString resource for an entry in the option menu.
- You can choose not to have these tips displayed by turning off the “Show tips on startup” option found on the RapidApp card in the RapidApp Preferences dialog accessed from the File menu.
- You can choose to have the startup panel dismissed automatically by turning on the “Auto-dismiss start screen” option found on the RapidApp card in the RapidApp Preferences dialog, accessed from the File menu.
- To display an image in a file, you can drag the pixmap or bitmap file from the desktop into labels, buttons, and so on. This is equivalent to setting the widget’s labelPixmap resource to that pixmap.
- You can drag Inventor data files directly onto Inventor viewers to view the image.
- RapidApp’s code ability to merge changes made to source code is greatly enhanced if you maintain blank lines between code generated by RapidApp and code entered by you.
- You can add classes you have created back to the RapidApp palette. See the RapidApp User’s Guide for details.
- You can create applications or libraries using RapidApp. Specify your library as the Library name on the Project card in the RapidApp Preferences dialog. All non-top-level window classes will be placed in your library so your classes can be shared with others.
- An Application class derived from **VkApp** is a convenient way to support command line options, application-level resources, and application state information. To have RapidApp create an Application class, assign a name in the VkApp subclass field on the Project card in the Preferences dialog.

- If you strictly follow the model of not modifying the base classes created by RapidApp, or consistently stay inside the editable code blocks, you may want to set the options on the Merge Options card in the Preferences dialog to “OverWrite” these files. In this mode, RapidApp does not attempt to merge changes to the base class files, thus saving time and reducing the potential for any conflicts.
- If you attempt to display many pixmaps that use large numbers of colors, you could run out of colors in the colormap. RapidApp attempts to match the desired colors to the closest available colors currently in use, but the results may be unexpected. You may want to reduce the number of colors a pixmap uses before loading it into RapidApp.
- RapidApp’s file format can only support pixmaps with 91 or fewer colors. Pixmaps that use more than 91 colors cannot be imported into RapidApp.
- RapidApp follows the ViewKit convention of adding leading underscores to all protected and private data members in the generated code. This helps avoid name collisions, allows you to add access methods that have the same name (without the leading underscore), and makes it easy to locate and identify private and protected data members in class code.
- If you plan to port your application to other Unix-based platforms, check the non-SGI Platform toggle on the Code Style card in the Preferences dialog. This produces a more portable *Makefile*. If you want to compile your application on other platforms, you also want to avoid SGI-specific widgets like the SgThumbwheel, SgDial, SgFinder, SgSpringBox and so on.
- The code generated by RapidApp is based on ViewKit and Inventor, which are available on many Unix-based platforms. ViewKit is distributed by Integrated Computer Solutions. Contact them at 617-621-0060 for details. Inventor is distributed by Template Graphics, who can be reached at (619) 457-5359.
- For classes that are to be reused by others, it is often useful to have default resources in the class rather than in an *app-defaults* file. If you set the “Place resources in classes” option on the Code Style card in the Preferences dialog, resources are associated with a class, using the ViewKit **setDefaultResources()** mechanism.
- You can change most widgets from one type to another by editing the class name in the instance header area. There are many ways to exploit this feature. For example, it is inherently difficult to add children to the paned window widget. You could, however, create a row column widget and add children in order. Once the children have been added, select the string “XmRowColumn” in the class name field of the header and type “XmPanedWindow.” After pressing <Return>, the row column widget is converted to a paned window widget.

- When modifying code generated by RapidApp, it's best to limit your changes to the areas that begin with

```
//--- Start editable code block
```

and end with

```
//--- End editable code block
```

Keeping your changes inside these markers allows RapidApp to merge changes easily.

- Setting the UI class option on the Merge Options card in the Preferences dialog allows you to control what files are merged when code is updated. For best results, limit your changes to derived classes to the areas that begin with

```
//--- Start user code block
```

and end with

```
//--- End user code block
```

If you limit changes to derived classes, you can configure RapidApp to "Overwrite" Base classes. This speeds up code generation and reduces the possibility of merge problems.

- Currently, the best way to use RapidApp with a configuration management system is to check out ALL files in the RapidApp project directory before working with RapidApp. Be sure to include the file named *RapidAppData.rap*, as well as the various auxiliary files that may be updated when you make changes using RapidApp. For large systems involving teams of programmers, use RapidApp to create libraries of components to organize responsibilities so that each program has a unique RapidApp project directory.
- To get the SGI "Percent Done" indicator, create a Scale widget and set the slanted resource to True, the slidingMode resource to XmTHERMOMETER, and the sliderVisual resource to XmFLAT_FOREGROUND. he slanted resource to True, the slidingMode resource to
- Before importing new components into RapidApp, it is useful to test them using the Component Tester application. This application is installed as part of RapidApp, and also can be found in the directory */usr/share/src/RapidApp/ComponentTester*. If you can load and interact with the component using this application, the component should work in RapidApp. If you have problems, run the source code for Component Tester under a debugger to find the problem.

- RapidApp forces some user interface elements to be classes. These include all entries in the Windows palette and the direct child of a SimpleWindow, VkWindow, or VkDialog. Unless you specify a name for these classes, RapidApp provides a generated name, which may be undesirable. You can select any of these elements and make them a class providing a name you prefer.
- You can add help to your application without writing any code. By default, a simple help library, *libvkhhelp*, is linked with your program. To use this system, set the resource:

```
*helpAuthorMode: true
```

Now, run your application and try the help menu items. A resource string is printed to standard output, which you can use as a resource in your application defaults file. For example, if asking for help produces a token like “application.form.label,” you can add the following to your application defaults file:

```
application.form.label.helpText: Help on this label
```

Once added (and after restarting the application), when you ask for help on this item, a dialog appears with the text “Help on this label.”

For a far more sophisticated help system, consider using the SGI Help System. See the SGI Indigo Magic Desktop Integration Guide for details.

- You can cycle through all widgets in your interface by pressing **<shift>** while clicking the right mouse button over the interface.
- The row column container provides an easy way to create a simple column of labeled fields:

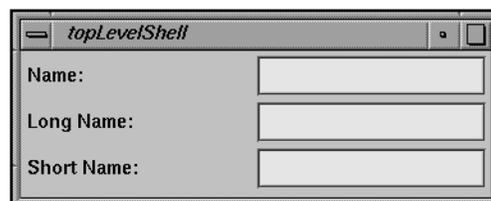


Figure C-1 Example of a Row Column Container

- Example programs built using RapidApp can be found under the directory `/usr/share/src/RapidApp`.
- By default, RapidApp creates a base class and a derived class for every class you define. RapidApp places all user interface code in the base class. You can implement virtual functions in the derived class to add behavior to the class. The approach separates the implementation of the user interface from the semantic behavior of the class, which is often a desirable characteristic.

If you do not wish to take advantage of this feature, toggle off the Split UI/Derived option found in the class header area when a class is selected in Class Edit mode. To globally change this behavior, go to the Code Style card in the Preferences dialog and unset the “Split classes into UI/Derived” option.

- You can edit the names and files associated with classes from within Class Edit mode. Enter class edit mode, select the class to edit, and issue the Select Parent command until you reach the top of the widget hierarchy. At that point, RapidApp’s header area changes to display information about the class, which can be edited.
- RapidApp allows you to create your own classes and add them to a palette. You can add behavior to these classes that is presented as resources in the RapidApp resource editor. See the *RapidApp User’s Guide* for details.
- When defining a class, you can choose from a set of base classes. The exact set depends on the type of class being created. You can also extend the set of base classes that can be used. See the *RapidApp User’s Guide* for details.

Source Code for the Calculator Application

This appendix lists and discusses some of the source files for the simple calculator application built in Chapter 1, “RapidApp Tour.” The version of the calculator program in this appendix includes the **Calculator** component created in “Creating Components.”

The Calculator main.C File

The body of any program generated by RapidApp is very simple. Example D-1 lists the *main.C* file for the calculator application.

Example D-1 Calculator *main.C* File

```

////////////////////////////////////
// This is a driver ViewKit program generated by RapidApp 1.2

//
// This program instantiates a ViewKit VkApp object and creates
// any main window objects that are meant to be shown at startup.
// Although editable code blocks are provided, there should rarely.
// be any reason to modify this file. Make application-specific
// changes in the classes created by the main window classes
// You can add also additional initialization in subclasses of VkApp
////////////////////////////////////
#include <Vk/VkApp.h>

// Headers for window classes used in this program

#include "CalcWindow.h"

//---- Start editable code block: headers and declarations

//---- End editable code block: headers and declarations

void main ( int argc, char **argv )
{

```

```
//---- Start editable code block: main initialization

//---- End editable code block: main initialization

VkApp      *app;

// Create an application object

app = new VkApp("Calculator", &argc, argv);

//---- Start editable code block: post init

//---- End editable code block: post init

// Create the top level windows

VkSimpleWindow *calcWindow = new CalcWindow("calcWindow");
calcWindow->show();

//---- Start editable code block: event loop

//---- End editable code block: event loop

app->run ();

}
```

This file simply instantiates an IRIS ViewKit **VkApp** class, 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 D-2 lists the **CalcWindowMainWindow** header file, and Example D-3 lists the **CalcWindowMainWindow** source file.

Example D-2 The Calculator *CalcWindowMainWindow.h* File

```
////////////////////////////////////
//
// Header file for CalcWindow
//
// This class is a subclass of VkSimpleWindow
//
// Normally, very little in this file should need to be changed.
// Create/add/modify menus using RapidApp.
//
// Restrict changes to those sections between
// the "//--- Start/End editable code block" markers
// Doing so will allow you to make changes using RapidApp
// without losing any changes you may have made manually
//
////////////////////////////////////
#ifndef CALCWINDOW_H
#define CALCWINDOW_H
#include <Vk/VkSimpleWindow.h>

//---- Start editable code block: headers and declarations

//---- End editable code block: headers and declarations

//---- CalcWindow class declaration

class CalcWindow: public VkSimpleWindow {

public:

    CalcWindow( const char * name,
                ArgList args = NULL,
                Cardinal argCount = 0 );
    ~CalcWindow();
    const char *className();
    virtual Boolean okToQuit();

//---- Start editable code block: CalcWindow public

//---- End editable code block: CalcWindow public
```

```
protected:

    // Classes created by this class
    class Calculator *_calculator;

    // Widgets created by this class

    //---- Start editable code block: CalcWindow protected

    //---- End editable code block: CalcWindow protected

private:

    static String _defaultCalcWindowResources[];

    //---- Start editable code block: CalcWindow private

    //---- End editable code block: CalcWindow private

};
//---- Start editable code block: End of generated code

//---- End editable code block: End of generated code
#endif
```

Example D-3 The Calculator *CalcWindowMainWindow.C* File

```
////////////////////////////////////
//
// Source file for CalcWindow
//
// This class is a subclass of VkSimpleWindow
//
//
// Normally, very little in this file should need to be changed.
// Create/add/modify menus using RapidApp.
//
// Try to restrict any changes to the bodies of functions
// corresponding to menu items, the constructor and destructor.
//
// Restrict changes to those sections between
// the "//--- Start/End editable code block" markers
//
// Doing so will allow you to make changes using RapidApp
// 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 RapidApp
////////////////////////////////////
#include "CalcWindow.h"

#include <Vk/VkApp.h>
#include <Vk/VkResource.h>

// Externally defined classes referenced by this class:

#include "Calculator.h"

extern void VkUnimplemented ( Widget, const char * );

//---- Start editable code block: headers and declarations

//---- End editable code block: headers and declarations
```

```

// These are default resources for widgets in objects of this class
// All resources will be prepended by *<name> at instantiation,
// where <name> is the name of the specific instance, as well as the
// name of the baseWidget. These are only defaults, and may be overridden
// in a resource file by providing a more specific resource name

String CalcWindow::_defaultCalcWindowResources[] = {
    "*title: Calculator",
    (char*)NULL
};

//---- Class declaration

CalcWindow::CalcWindow ( const char *name,
                        ArgList args,
                        Cardinal argCount) :
    VkSimpleWindow ( name, args, argCount )
{
    // Load any class-default resources for this object

    setDefaultResources ( baseWidget(), _defaultCalcWindowResources );

    // Create the view component contained by this window

    _calculator = new Calculator ( "calculator",mainWindowWidget() );

    XtVaSetValues ( _calculator->baseWidget(),
                    XmNwidth, 201,
                    XmNheight, 141,
                    (XtPointer) NULL );

    // Add the component as the main view

    addView ( _calculator );

    //---- Start editable code block: CalcWindow constructor

    //---- End editable code block: CalcWindow constructor

```

```

} // End Constructor

CalcWindow::~CalcWindow()
{
    delete _calculator;
    //---- Start editable code block: CalcWindow destructor

    //---- End editable code block: CalcWindow destructor
} // End destructor

const char *CalcWindow::className()
{
    return ("CalcWindow");
} // End className()

Boolean CalcWindow::okToQuit()
{
    //---- Start editable code block: CalcWindow 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 editable code block: CalcWindow okToQuit
} // End okToQuit()

//---- Start editable code block: End of generated code

//---- End editable code block: End of generated code

```

Note the “End Generated Code Section” markers. If you need to modify this class, 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 the program exits, 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 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 Class

Calculator is the user-defined class you created in RapidApp. RapidApp automatically places most of the user interface code in the base class, **CalculatorUI**, so the **Calculator** class itself is very simple. The class header, shown in Example D-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 D-4 The Calculator *Calculator.h* File

```
////////////////////////////////////
//
// Header file for Calculator
//
// This file is generated by RapidApp 1.2
//
// This class is derived from CalculatorUI which
// implements the user interface created in
// RapidApp. This class contains virtual
// functions that are called from the user interface.
//
// When you modify this header file, limit your changes to those
// areas between the "//--- Start/End editable code block" markers
//
// This will allow RapidApp 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"
//---- Start editable code block: headers and declarations

//---- End editable code block: headers and declarations

//---- Calculator class declaration

class Calculator : public CalculatorUI
{
public:
    Calculator(const char *, Widget);
    Calculator(const char *);
    ~Calculator();
    const char * className();

    static VkComponent *CreateCalculator( const char *name, Widget parent );
};
```

```
    //---- Start editable code block: Calculator public

    //---- End editable code block: Calculator public

protected:

    // These functions will be called as a result of callbacks
    // registered in CalculatorUI

    virtual void add ( Widget, XtPointer );

    //---- Start editable code block: Calculator protected

    //---- End editable code block: Calculator protected

private:

    static void* RegisterCalculatorInterface();

    //---- Start editable code block: Calculator private

    //---- End editable code block: Calculator 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 D-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 D-5 The Calculator *Calculator.C* File

```
////////////////////////////////////
//
// Source file for Calculator
//
// This file is generated by RapidApp 1.2
//
// This class is derived from CalculatorUI which
// implements the user interface created in
// RapidApp. This class contains virtual
// functions that are called from the user interface.
//
// When you modify this source, limit your changes to
// modifying the sections between the
// "///--- Start/End editable code block" markers
//
// This will allow RapidApp 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 <Xm/BulletinB.h>
#include <Xm/Label.h>
#include <Xm/PushButton.h>
#include <Xm/Separator.h>
#include <Xm/TextF.h>
#include <Vk/VkResource.h>
#include <Vk/VkSimpleWindow.h>

extern void VkUnimplemented(Widget, const char *);
```

```

////////////////////////////////////
// The following non-container elements are created by CalculatorUI and are
// available as protected data members inherited by this class
//
// XmPushButton          _pushbutton
// XmTextField           _result
// XmSeparator           _separator
// XmLabel               _label
// XmTextField           _value2
// XmTextField           _value1
//
////////////////////////////////////

//---- Start editable code block: headers and declarations

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

//---- End editable code block: headers and declarations

//---- 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

    //---- Start editable code block: Calculator constructor

    //---- End editable code block: Calculator constructor

} // 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

    //---- Start editable code block: Calculator constructor 2

    //---- End editable code block: Calculator constructor 2

} // 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.

    //---- Start editable code block: Calculator destructor

    //---- End editable code block: Calculator destructor

}

const char * Calculator::className() // classname
{
    return ("Calculator");
} // End className()

void Calculator::add ( Widget w, XtPointer callData )
{
    //---- Start editable code block: Calculator add

    XmPushButtonCallbackStruct *cbs = (XmPushButtonCallbackStruct*) callData;

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

```

```

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

int x, y;

x = atoi(XmTextFieldGetString(_value1));
y = atoi(XmTextFieldGetString(_value2));

XmTextFieldSetString(_result, (char *)VkFormat("%d", x+y));

//---- End editable code block: Calculator add

} // End Calculator::add()

////////////////////////////////////
// static creation function, for importing class into rapidapp
// or dynamically loading, using VkComponent::loadComponent
////////////////////////////////////

VkComponent *Calculator::CreateCalculator( const char *name, Widget parent )
{
    VkComponent *obj = new Calculator ( name, parent );
    return ( obj );
} // End CreateCalculator

////////////////////////////////////
// Function for accessing a description of the dynamic interface
// to this class.
////////////////////////////////////

void *Calculator::RegisterCalculatorInterface()

{
    // This structure registers information about this class
    // that allows RapidApp to create and manipulate an instance.
    // Each entry provides a resource name that will appear in the
    // resource manager palette when an instance of this class is
    // selected, the name of the member function as a string,
    // the type of the single argument to this function, and an.
    // optional argument indicating the class that defines this function.

```

```

// All functions must have the form
//
//     void memberFunction(Type);
//
// where "Type" is one of:
//     const char *    (Use XmRString)
//     Boolean         (Use XmRBoolean)
//     int             (Use XmRInt)
//     float           (Use XmRFloat)
//     No argument     (Use VkRNoArg)
//     A filename      (Use VkRFilename)
//     An enumeration (Use "Enumeration:ClassName:TYPE: VALUE1, VALUE2,
VALUE3")

static VkCallbackObject::InterfaceMap map[] = {
//---- Start editable code block: CalculatorUI resource table

    // { "resourceName", "setAttribute", XmRString},
//---- End editable code block: CalculatorUI resource table
    { NULL }, // MUST be NULL terminated
};

return map;
} // End RegisterCalculatorInterface()

//---- End of generated code

//---- Start editable code block: End of generated code

//---- End editable code block: End of generated code

```

The Calculator Resource File

The *Calculator* file contains the default values for various X resources used by the calculator application. Example D-6 lists the calculator resource file. You typically shouldn't need to edit this file.

Example D-6 The Calculator Resource File

```
!  
! Generated by Silicon Graphic's RapidApp.  
!  
!  
! RapidApp 1.2.  
!  
!  
!  
!Activate schemes and sgi mode by default  
!  
Calculator*useSchemes: all  
Calculator*sgiMode: true  
!  
!SGI Style guide specifies explicit focus within applications  
!  
Calculator*keyboardFocusPolicy: explicit  
  
  
Calculator*calcWindow.title: Calculator  
Calculator*label.labelString: +  
Calculator*pushbutton.labelString: =  
Calculator*showHelp: True  
  
!!---- Start editable code block: app-defaults  
  
Calculator*applicationVersionString: Version 1.0  
  
Created by RapidApp(tm)  
  
!!---- End editable code block: app-defaults
```

Makefile

The *Makefile* follows Silicon Graphics conventions. It also uses a few simple conventions that make it easier to maintain from within RapidApp.

Example D-7 The Calculator *Makefile* File

```
#!/smake
#
# Makefile for calculator
# Generated by RapidApp 1.2
#
# This makefile follows various conventions used by SGI makefiles
# See the RapidApp User's Guide for more information
# This makefile supports most common default rules, including:
# make (or make all): build the application or library
# make install:      install the application or library on the local machine
# make image:       create "inst" images for distribution
# make clean:       remove .o's, core, etc.
# make clobber:     make clean + remove the target application.
# You should be able to customize this Makefile by editing
# only the section between the ##---- markers below.
# Specify additional files, compiler flags, libraries
# by changing the appropriate variables
include $(ROOT)/usr/include/make/commondefs

##---- Start editable code block: definitions

#####
#####
# Modify the following variables to customize this makefile
#####
#####
#
# Local Definitions
#

# Directory in which inst images are placed

IMAGEDIR= images

# Add Additional libraries to USERLIBS:

USERLIBS=
```

```
# While developing, leave OPTIMIZER set to -g.
# For delivery, change to -O2

OPTIMIZER= -g
#
# Add any files added outside the builder here
#

USERFILES =

#
# Add compiler flags here
#

USERFLAGS =

##---- End editable code block: definitions

# 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 -lhelpmsg
# switches to the full IRIS Insight help system
#

HELPLIB= -lvkhelp

# Standard ViewKit header and libraries

VIEWKITFLAGS= -I$(ROOT)/usr/include/Vk
TOOLTALKLIBS=
NETLS=

EZLIB =
VIEWKITLIBS= $(TOOLTALKLIBS) $(EZLIB) -lvk -lvkSGI $(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. -I$(ROOT)/usr/include $(SAFLAG) $(WOFF)
$(VIEWKITFLAGS) $(USERFLAGS)

LLDLIBS = -L$(ROOT)/usr/lib $(USERLIBS) $(COMPONENTLIBS) $(VIEWKITLIBS)
$(GLLIBS) -lXm -lXt -lX11 -lgen

# SGI makefiles don't recognize all C++ suffixes, so set up
# the one being used here.

CXXO3=$(CXXO2:.C=.o)
CXXOALL=$(CXXO3)

#
# Source Files generated by RapidApp. If files are added
# manually, add them to USERFILES
#
BUILDERFILES = main.C\p                CalcWindow.C\p
Calculator.C\p                CalculatorUI.C\p
unimplemented.C\p                $(NULL)

C++FILES = $(BUILDERFILES) $(USERFILES)

#
# The program being built
#

TARGETS=calculator
APPDEFAULTS=Calculator
default all: $(TARGETS)

$(TARGETS): $(OBJECTS)
    $(C++F) $(OPTIMIZER) $(OBJECTS) $(LDFLAGS) -o $@
#
# These flags instruct the compiler to output
# analysis information for cvstatic
# Uncomment to enable
# Be sure to also disable smake if cvstatic is used

```

```

#SADIR= Motif.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
    $(INSTALL) -F /usr/lib/images Calculator.icon

#
# To create inst images, do 'make image'
# An image subdirectory should already exist
#

$(IMAGEDIR):
    @mkdir $(IMAGEDIR)
image: $(TARGETS) $(IMAGEDIR)
    /usr/sbin/gendist -rbase / -sbase `pwd` -idb calculator.idb \\p
-spec calculator.spec \\p          -dist
/usr/share/src/RapidApp/Calculator/Motif/images -all

include $(COMMONRULES)

```

You should rarely need to modify most of the *Makefile*. However, the editable code block area is intended to be changed. You can edit variables declared in that area to add files, libraries, and change other aspects of the *Makefile*.

- The variable `IMAGEDIR` controls the location at which installable images are generated. The default is a subdirectory of the current directory called *images*.
- 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.

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 *constraint resources*.

base widget

The root widget of a component's single-rooted widget subtree. Components typically use a container widget as the root of the subtree; all other widgets are descendants of this widget. See also *components*.

cascade buttons

Push buttons that, when you click 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++ gui class and can contain several other components and/or widgets. See also *interface elements*, *widgets*, and *user-defined components*.

constraint resources

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

Widgets 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 data outside of the main window. See also *main windows*.

elements

See *interface elements*.

interface elements

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 you toggle 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*.

support windows

A type of window that typically contains a control panel or tool palette that operates directly on data in a main or co-primary window. See also *co-primary windows* and *main windows*.

VkUnimplemented()

This function is placed in user-defined callback functions and is used as follows:

- While running and testing an application and when a callback is encountered, this function prints a message telling you that the callback has been called.
- While debugging an application and when a callback is encountered, this function is called. This is useful when you have an undefined callback function.

widgets

Interface components that are part of the IRIS IM toolkit. See also *interface elements* and *IRIS IM*.

Index

A

- aborting interface element creation, 28
- activateCallback resource, 38
- Application.icon* file, 46
- application resource files, 14
- applications
 - building, 44
 - code
 - adding, 50-54
 - editable code blocks, 50, 52, 54
 - editing, 50-54
 - editor, 54
 - EZ convenience functions, 54
 - generation, 49-??
 - merging, 50
 - restricted areas, 50, 52, 54
 - rules for RapidApp, 50
 - user input areas, 50, 52, 54
 - compiling, 44
 - creating, 43-47
 - installable images, 46
 - debugging, 51
 - development cycle, 43-47
 - development model, 47-48
 - files, list of, 55
 - functionality
 - adding, 50-54

icons

- desktop, 45
- minimized window, 46
- saving files, 42
- testing, 44, 52

attachment icons, 303

B

- base widget, 303
- blocking, modal dialogs, 98
- block merge, 58
- “Browse Source” selection (in Project menu), 45
- “Build Application” selection (in Project menu), 44
- .buildersource* directory, 60-61
- building. *See* compiling
- Build Manager, 44
- Bulletin Board, 73-74
- buttons
 - specifying behavior, 38

C

- callback functions, 38
- child elements, 303
- class name, 19

- code
 - adding, 50-54
 - adding interactively, 52
 - editable code blocks, 50, 52, 54
 - editing, 50-54
 - editor, 54
 - EZ convenience functions, 54
 - generation, 49-??
 - include files, 51
 - including outside files, 51
 - management, 55-61
 - merging, 50, 58-61
 - restricted areas, 50, 52, 54
 - rules for RapidApp, 50
 - setting flags, 51
 - user input areas, 50, 52, 54
- “Color by Depth” selection (in View menu), 40
- colormaps, 14
- colors
 - changing, 51
- color schemes, 51
- compiling, 44
 - component libraries, 126
- Component Importer dialog, 130-132
- component libraries, 123-150
 - creating, 123-128
 - configuring RapidApp, 124
 - default resource, 124
 - generating, 125-127
 - installable images, creating, 136
 - installing, 127
 - loading components into RapidApp, 128-150
 - packaging for installation, 127-128
 - testing component, 147-148
- components, 12, 47-48, 104-111, 303
 - See also* interface elements
 - creating, 105-108
 - derived class, 105
 - deleting from custom palettes, 135
 - loading from component libraries, 128-150
 - resources, adding to custom, 136-141
 - testing custom components, 147-148
- Component Tester, 147-148
- constraint resources, 39
- constraints, 73, 303
 - See also* resources
- containers, 72-92, 303
 - Bulletin Board, 73-74
 - child elements
 - moving, 30-35
 - moving to another container, 35
 - reparenting, 35
 - repositioning, 30-35
 - constraint resources, 39
 - constraints, 73
 - creating within containers, 30
 - Drawing Area, 90
 - Form, 82-86
 - Frame, 89-90
 - HPaned Window, 86-88
 - moving, 29
 - Paned Window, 86-88
 - Radio Box, 89
 - resizing, 29
 - RowColumn, 88-89
 - Rubber Board, 74-78
 - Scrolled Window, 90
 - Spring Box, 79-82
 - Tabbed Deck, 91-92
 - Visual Drawing, 90

co-primary windows, 69-70, 304
See also windows

copying interface elements, 28

“Copy” selection (in Edit menu), 28, 35

creating

- applications, 43-47
- component libraries, 123-128
- components, 105-108
- containers within containers, 30
- icons
 - desktop, 45
 - minimized window, 46
- installable images, 46
- interface elements, 25-28
 - aborting, 28
- menu bars, 93
- menu items, 95-96
- menu panes, 93-94

custom-designed windows, 70

custom dialogs, 99-104

custom palettes, 128, 129-132, 134-135

- deleting components, 135

“Cut” selection (in Edit menu), 35, 36

cutting

- interface elements, 36

D

“Debug Application” selection (in Project menu), 51

Debugger, 51

debugging, 51

“Delete” selection (in Edit menu), 36

deleting

- components from custom palettes, 135
- interface elements, 36
- menu items, 96
- menu panes, 94

desktop.ftr, 45

desktop icon, 45

- desktop tag, 45

desktop tag, 45

development cycle, 43-47

development model, 47-48

dialogs, 97-104

- custom, 99-104

Did you know? in startup screen, 16, 271

displaying menu panes, 94

“Down/Right” selection (in Edit menu), 34

Drawing Area, 90

dynamic resources, 39

E

editable code blocks, 50, 52, 54

“Edit Classes” selection (in Classes menu), 111

“Edit Files” selection (in Project menu), 54

editing

- resources, 37-40

Edit menu, 36

editor, 54

- setting up, 14

elements. *See* interface elements.

explicit focus mode, 24

extended resources, 40

EZ convenience functions, 54, 113-121

F

Fix and Continue, 52

fonts

- changing, 51

font schemes, 51

Form, 82-86

Frame, 89-90
functional code
 adding interactively, 52
 separate from interface code, 57-58, 106
functionality of interface elements, 37-40
functions
 EZ convenience, 54

G

“Generate C++” selection (in Project menu), 42, 49
generating code, 49-??
 component libraries, 125
GLDraw, 90
GLwMDrawingArea, 90
grid
 resolution, 31
 setting, 31
 turning off, 31
“Grow Widget” selection (in Edit menu), 34

H

help
 quick help, 19
hierarchy, viewing widget, 40
HPaned Window, 86-88

I

icon.fti file, 45
icons
 desktop, 45
 minimized window, 46
include files, 51

Indigo Magic Desktop environment
 look, 21
installable images, 46
 component libraries, 127-128, 136
installing
 component libraries, 127
 RapidApp, 13
instance header, 19
instance name, 19
interface
 saving, 42
interface code, separate from functional code, 57-58,
 106
interface description, RapidApp’s, 17-19
interface elements, 12, 304
 See also components, widgets
 aborting creation, 28
 behavior, 37-40
 colors, 51
 copying, 28
 creating, 25-28
 creation
 confining, 24
 cutting, 36
 deleting, 36
 fonts, 51
 functionality, 37-40
 hierarchy, viewing, 40
 locking on, 40
 minimum size, 25, 29
 modifying, 37-40
 moving, 29-36
 to another container, 35
 moving difficulties, 31
 naming, 36
 parents, selecting, 24
 pasting, 28
 preventing selection, 40
 reparenting, 35
 repositioning, 29-36

repositioning difficulties, 31
 resizing, 29-36
 difficulties, 34
 resizing difficulties, 31
 resources, 37-40
 selecting, 24
 difficulties, 40
 specifying behavior, 38
 IRIS IM, 304
 IRIS ViewKit, 47-48

K

"Keep Parent" selection (in View menu), 24
 keyboard accelerators, 65

L

LD_LIBRARY_PATH environment variable, 126
 Library Header Directory field (in Application Options dialog), 124
 locking on to an element, 40

M

main window, RapidApp's, 17-19
 main windows, 69-70, 304
 See also windows
 "Make Class" selection (in Classes menu), 105
 managing code, 55-61
map data member, 139
 menu bar, RapidApp, 18

menu bars, 92-94
 creating, 93
 creating menu panes, 93-94
 deleting menu panes, 94
 moving menu panes, 94
 standard application entries, 65
 menu items
 creating, 95-96
 deleting, 96
 moving, 96
 menu panes, 94-96
 creating, 93-94
 creating items in, 95-96
 deleting, 94
 deleting items from, 96
 displaying, 94
 moving, 94
 moving items in, 96
 menus, 92-96
 creating items in, 95-96
 deleting items from, 96
 displaying, 94
 menu bars, 92-94
 menu panes, 94-96
 moving items in, 96
 option menus, 96
 Merge Options card, 59
 "Merge Options" card (in Preferences dialog), 59
 merging code, 58-61
 minimized window icon, 46
 minimum size, interface elements, 25, 29
 model, developing applications, 47-48
 modifying interface elements, 37-40

moving
containers, 29
difficulties, 31
interface elements, 29-36
to another container, 35
menu items, 96
menu panes, 94
windows, 29

N

naming interface elements, 36
non-blocking, modal dialogs, 98
non-blocking, non-modal dialogs, 98

O

object-oriented components, 47-48
option menus, 96
overview, RapidApp, 11-12

P

palettes, 18
custom, 128, 129-132, 134-135
deleting components, 135
tabs, 19
Paned Window, 86-88
parent elements
selecting, 24
"Paste" selection (in Edit menu), 28, 35
pasting interface elements, 28
pointer focus mode, 24

Preferences dialog, 59
Merge Options card, 59
"Preferences" selection (in File menu), 42, 44
project directory
setting up, 13
pseudo-constraints, 39
push button
specifying behavior, 38

R

radio behavior, 304
Radio Box, 89
RapidApp
coding rules, 50
colormap, 14
development cycle, 43-47
development model, 47-48
installing, 13
main window, 17-19
overview, 11-12
setting up environment, 13
RapidApp startup screen, 15
reparenting interface elements, 35, 304
repositioning difficulties, 31
repositioning interface elements, 29-36
resizing
containers, 29
difficulties, 31, 34
interface elements, 29-36
windows, 29
resource editor, 19
resource files
setting up, 14

resources, 37-40, 304
 adding to custom components, 136-141
 callback functions, 38
 constraints, 39
 default, in component libraries, 124
 dynamic, 39
 editing, 37-40
 extended, 40

RowColumn, 88-89

Rubber Board, 74-78

S

“Save As” selection (in File menu), 42

saving
 application files, 42
 interface files, 42
 setting up, 42

Scrolled Window, 90

selecting difficulties, 40

setting up
 colormaps, 14
 editor, 14
 project directory, 13
 RapidApp, 13
 resource files, 14

SgHorzPanedWindow, 86-88

SgI look and feel, 21

SgRubberBoard, 74-78

SgSpringBox, 79-82

SgVisualDrawingArea, 90

shared libraries, generated for component libraries,
 126

Simple Windows, 65

snap grid, 31, 304

“Snap to Grid” selection (in View menu), 31

Software Packager, 46

Source View editor, 54

Spring Box, 79-82

starting RapidApp, 15

startup screen, 15
 setting preferences, 16

static analysis, 44

Static Analyzer, 44

support windows, 305

T

Tabbed Deck, 91-92

tabs, 19

templates, window, 70

text editor, 54

three-way merge, 59

tips
 in startup screen, 16, 271
 viewing, 16, 271

ToolTalk, 56

U

“UI” classes, 57-58, 106

“Up/Left” selection (in Edit menu), 34

user-defined components
See components.

V

viewing widget hierarchy, 40

Visual Drawing, 90

VkComponent class, 105

VkEZ, 113-121

VkMsgApp class, 56

VkMsgComponent class, 105
VkTabbedDeck, 91-92
VkUnimplemented(), 52
VkWindows, 65-69

W

widgets, 12, 305
 See also interface elements
 viewing hierarchy, 40
windows, 63-71
 co-primary, 69-70
 custom-designed, 70
 dialogs, 97-104
 custom, 99-104
 main, 69-70
 moving, 29
 resizing, 29
 selecting, 24
 Simple Windows, 65
 templates, 70
 user-defined, 70
 VkWindows, 65-69
WINEDITOR environment variable, 14

X

XmBulletinBoard, 73-74
XmDrawingArea, 90
XmForm, 82-86
XmFrame, 89-90
XmPanedWindow, 86-88
XmRowColumn, 88-89, 89
XmScrolledWindow, 90
XUSERFILESEARCHPATH environment
 variable, 14

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2590-004.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
 - On the Internet: techpubs@sgi.com
 - For UUCP mail (through any backbone site): *[your_site]!sgi!techpubs*
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389