



IBM ILOG Views

Controls V5.3

User's Manual

June 2009

© **Copyright International Business Machines Corporation 1987, 2009.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, Websphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further information see *<installdir>/license/notices.txt* in the installed product.

Table of Contents

Preface	About This Manual 20 What You Need to Know 20 Manual Organization 20 Notation 21 Typographic Conventions 21 Naming Conventions 21
Part I	Creating GUI Applications with IBM ILOG Views Studio 22
Chapter 1	Introducing the Gadgets Extension of IBM ILOG Views Studio 24 Loading the GUI Application and GUI Generation Plug-In 25 The Main Window 25 Buffer Windows 26 The Menu Bar 29 The Action Toolbar 31 The Editing Modes Toolbar 31 The Palettes Panel 33 Gadgets Palette 34 Menu Palette 36

Matrix Palette	36
Miscellaneous Palette	37
View Rectangles Palette	39
Gadgets Extension Commands	39
AddPanel	40
EditApplication	40
Generate	40
GenerateAll	41
GenerateApplication	41
GenerateMakeFile	41
GeneratePanelClass	42
GeneratePanelSubClass	42
InspectPanel	42
KillTestPanels	42
MakeDefaultApplication	43
NewApplication	43
NewGadgetBuffer	43
NewPanelClass	44
OpenApplication	44
RemoveAllAttachments	44
RemoveAttachments	44
RemovePanel	45
RemovePanelClass	45
SaveApplication	45
SaveApplicationAs	45
SelectAttachmentsMode	46
SelectFocusMode	46
SelectMatrixMode	46
SelectMenuMode	47
ShowAllTestPanels	47
ShowApplicationInspector	47

	ShowClassPalette	48
	ShowPanelClassInspector	48
	TestApplication	48
	TestDocument	48
	TestPanel	49
	Gadgets Extension Panels	49
	Application Inspector	49
	The Panel Class Inspector	51
	The Panel Instance Inspector	52
Chapter 2	Editing Gadget Panels	54
	Creating a New Panel.....	55
	Creating Gadget Objects.....	55
	Inspecting an Object	58
	Testing a Panel	59
	Using Active Mode.....	59
	Setting the Keyboard Focus in Panels	60
	Using the Attachments Mode	61
	Setting the Guides	62
	Attaching Objects to Guides.....	63
	Attachment Operations	64
	Editing Menus.....	66
	Menu Bars	66
	Pop-up Menus	69
	Toolbars.....	72
	Using Matrices	75
	Setting Up Matrix Items	75
	Extracting Matrix Items.....	76
	Inspecting Matrix Items	76
	Editing Spin Boxes	78
	Inserting a Spin Box.....	79
	Setting the Type of Spin Box Item	80

Chapter 3	Editing Applications	84
	The Application Buffer	84
	Application Description File	88
	Other Generated Files	89
	The Application Inspector	89
	The General Page	90
	The Options Page	91
	The Header and Source Pages	93
	The Script Page	94
	The Application Inspector Buttons	95
	Editing an Application	95
	The Panel Class Palette	96
	Panel Classes	97
	The Panel Class Inspector	98
	Panel Instances	102
	Testing an Application	111
Chapter 4	Using the Generated Code	112
	Building the Application	112
	Setting Up the Application Class	113
	Creating the First Panel Class	114
	Creating the Second Panel Class	118
	Generating the C++ Code	121
	FirstPanelClass Header File	121
	FirstPanelClass Source File	124
	MyApplication Header File	125
	MyApplication Source File	127
	Testing the Generated Application	129
	Extending the Generated Code	129
	Defining a Derived Class	129
	Using the Derived Class	130
	Defining Callbacks without Deriving Classes	133

Chapter 5	Customizing the Gadgets Extension of IBM ILOG Views Studio	136
	Configuration Options for the Gadgets Extension	136
Chapter 6	Extending IBM ILOG Views Studio	142
	Extending IBM ILOG Views Studio Components	142
	Defining a New Command	143
	Defining a New Panel.	144
	IBM ILOG Views Studio Messages	144
	Defining a New Buffer	145
	Defining a New Editing Mode.	146
	The Class IlvStExtension	147
	Integrating your Own Graphic Objects	151
	Defining a New Command to Add an Object	151
	Adding the Include File and Library File of a New Class to the Generated Code	152
	Customizing the Palettes Panel	153
	Defining and Integrating an Inspector Panel	154
	Extending IBM ILOG Views Studio: An Example	156
	Defining a New Buffer Class	156
	Defining a New Command	158
	Defining a New Panel.	159
	Providing Container Information.	161
	Registering Callbacks	162
Chapter 7	Using Inspector Classes	164
	What Is an Inspector?	164
	Components of an Inspector Panel	165
	Preconditions and Validators	179
	Editors	181
	Defining a New Inspector Panel	182
	Example	182
	Creating the Color Combo Box Inspector Panel	183

Part II	IBM ILOG Views Gadgets	194
Chapter 8	Introducing IBM ILOG Views Gadgets	196
	Gadgets Main Features	196
	Gadgets in a Snapshot	197
	Menus	197
	Common Gadgets	197
	Matrices	198
	Gadgets Libraries	198
	Code Sample	200
Chapter 9	Understanding Gadgets	202
	Gadget Holders	202
	List of Available Gadget Holders	203
	Handling Events	204
	Focus Management	204
	Gadgets Attachments	206
	Common Gadget Properties	208
	Gadget Appearance	208
	Associating a Callback with a Gadget	210
	Localizing a Gadget	211
	Associating a Mnemonic with a Gadget Label	212
	Setting Tooltips	212
	Gadget Resources	213
	Gadgets Look and Feel	217
	Using the Default Look and Feel	218
	Using Several Look and Feel	219
	Dynamic Loading of Look and Feel	220
	Changing the Look and Feel Dynamically	221
	Using the Windows XP Look and Feel	222
Chapter 10	Dialogs	224

	Predefined Dialog Boxes	225
	IlvMessageDialog	225
	IlvQuestionDialog	226
	IlvErrorDialog	226
	IlvWarner	227
	IlvInformationDialog	227
	IlvFileSelector	227
	IlvPromptString	228
	IlvFontSelector	229
	IlvColorSelector	229
	Creating Your Own Dialog Box	230
	Showing and Hiding Dialog Boxes	231
	Setting a Default Button	231
Chapter 11	Using Common Gadgets	232
	Using IlvArrowButton	233
	Using IlvButton	233
	Displaying a Bitmap in a Button	234
	Displaying the Button Frame	234
	Associating a Mnemonic with a Button	234
	Event Handling and Callbacks	234
	Using IlvComboBox and IlvScrolledComboBox	235
	Setting a Combo Box as Noneditable	235
	Setting and Retrieving Items	235
	Changing or Retrieving the Selection	236
	Using Large Lists	236
	Setting the Number of Visible Items	236
	Localizing Combo Boxes	236
	Event Handling and Callbacks	236
	Using IlvDateField	237
	Formatting a Date	237
	Setting and Retrieving a Date Value	238

Year 2000 Management	238
Using IlvFrame	239
Associating a Mnemonic with a Frame	239
Using IlvMessageLabel	239
Associating a Bitmap with a Message Label	240
Making the Message Label Opaque	241
Laying Out the Message Label	241
Localizing a Message Label.	242
Associating a Mnemonic	242
Using IlvNotebook	242
Customizing Notebook Tabs	242
Handling Notebook Pages	244
Event Handling and Callbacks.	247
Using IlvNumberField	247
Selecting an Editing Mode	248
Choosing a Format.	248
Defining a Range of Values	249
Setting and Retrieving a Value.	249
Specifying the Thousand Separator.	249
Specifying the Decimal Point Character.	250
Event Handling and Callbacks.	250
Using IlvOptionsMenu	250
Setting and Retrieving Items	251
Changing and Retrieving the Selected Item.	251
Localizing Option Menus	251
Event Handling and Callbacks.	251
Using IlvPasswordTextField	251
Using IlvScrollBar	252
Setting the Scrollbar Values.	252
Setting the Scrollbar Orientation	252
Event Handling and Callbacks.	253

Using IlvSlider	253
Setting the Slider Values	254
Setting the Slider Orientation	254
Setting the Thumb Orientation	254
Event Handling and Callbacks	255
Using IlvSpinBox	255
Adding and Removing Fields to a Spin Box	256
Working with Text Fields	257
Working with Numeric Fields	257
Event Handling and Callbacks	258
Using IlvStringList	258
Manipulating String List Items	258
Customizing the Appearance of String List Items	259
Displaying Tooltips	260
Localizing String List Items	261
Handling Events and Callbacks	261
Using IlvText	262
Setting and Retrieving Text	263
Event Handling	263
Using IlvTextField	264
Aligning Text	265
Setting and Retrieving Text	265
Localizing a Text Field	265
Limiting the Number of Characters	265
Event Handling and Callbacks	266
Keyboard Shortcuts	267
Using IlvToggle	267
Changing the State and Color of a Toggle Button	268
Toggle and Radio Button Styles	268
Displaying a Bitmap on a Toggle Button	269
Aligning and Positioning the Label	269

	Changing the Size of the State Marker	269
	Localizing a Toggle Button	269
	Associating a Mnemonic with a Toggle Button	269
	Handling Events and Callbacks	270
	Grouping Toggle Buttons in a Selector	270
	Using IlvTreeGadget	271
	Changing the Tree Hierarchy	272
	Navigating Through a Tree Hierarchy	273
	Changing the Characteristic of an Item	273
	Expanding and Collapsing a Gadget Item	273
	Changing the Look of the Tree Gadget Hierarchy	274
	Event Handling and Callbacks	275
Chapter 12	Gadget Items	278
	Introducing Gadget Items	278
	Using Gadget Items	279
	Creating a Gadget Item	280
	Setting a Label	280
	Setting a Picture	281
	Specifying the Layout of a Gadget Item	281
	Nonsensitive Gadget Items	282
	Dynamic Types	282
	Using Palettes with Gadget Items	283
	Drawing a Gadget Item	283
	Gadget Item Holders	283
	Gadget Item Features	284
	Finding Gadget Items	284
	Redrawing Gadget Items	284
	Creating Gadget Items	285
	Editing Gadget Items	285
	Dragging and Dropping Gadget Items	286
	List Gadget Item Holders	287

	Modifying a List	287
	Accessing Items	288
	Sorting a List	289
Chapter 13	Menu, Menu Bars, and Toolbars	290
	Introducing Menus, Menu Bars, and Toolbars	290
	Menus and Menu Items	291
	Using IlvAbstractMenu	291
	Using IlvMenuItem	292
	Pop-up Menus	294
	Aligning Item Labels in a Pop-up Menu	295
	Using Tear-Off Menus	296
	Using the Open Menu Callback	296
	Using Checked Menu Items	296
	Using Stand-alone Menus	297
	Using Tooltips in a Pop-Up Menu	297
	Menu Bars and Toolbars	298
	Using IlvAbstractBar	298
	Using IlvMenuBar and IlvToolBar	300
Chapter 14	Matrices	302
	Introducing Matrices	302
	Using IlvAbstractMatrix	303
	Subclassing IlvAbstractMatrix	303
	Drawing Items Over Multiple Cells	304
	Setting Fixed Rows and Columns	304
	Handling Events	305
	Using IlvMatrix	305
	Handling Columns and Rows	306
	Handling Matrix Items	307
	Handling Events	312
	Using Gadget Items in a Matrix	315

	Using IlvSheet	316
	Using IlvHierarchicalSheet	317
	Changing the Tree Hierarchy	317
	Navigating through a Tree Hierarchy	318
	Changing the Characteristic of a Tree Item	318
	Expanding and Collapsing a Gadget Item	318
	Changing the Look of the Tree Gadget Hierarchy	318
	Event Handling and Callbacks	319
Chapter 15	Panes	320
	Introducing Panes	320
	Creating Panes	323
	Creating a Graphic Pane	323
	Creating a View Pane	323
	Showing or Hiding a Pane	324
	Adding Panes to Paned Containers	324
	Creating a Paned Container	324
	Modifying the Layout of a Paned Container	325
	Retrieving Panes	325
	Encapsulating a Paned Container in a View Pane	325
	Resizing Panes	326
	Setting the Resize Mode and the Minimum Size of a Pane	326
	Resizing Panes With Sliders	327
Chapter 16	Docking Panes and Containers	330
	Introducing Docking Panes and Dockable Containers	331
	Creating Docking Panes	333
	Creating Orthogonal Dockable Containers	335
	Controlling Docking Operations	338
	Connecting an Instance of the IlvDockable Class to a Pane	338
	Docking and Undocking a Pane	338
	Filtering Docking Operations	339

	Using Docking Bars	339
	Using the IlvAbstractBarPane Class	340
	Customizing Docking Bars	341
	Building a Standard Application With Docking Panes	342
	Defining a Standard Layout	342
	Using the IlvDockableMainWindow Class	345
Chapter 17	View Frames	350
	Introducing View Frames	350
	Creating a Desktop with View Frames	351
	Creating a Desktop	351
	Creating View Frames	352
	Managing View Frames	352
	Creating a Client View	353
	Changing the Title Bar	353
	Changing the View Frame Menu	353
	Minimizing, Maximizing, and Restoring View Frames	354
	Normal View Frames	355
	Minimized View Frames	355
	Maximized View Frames	356
	Closing View Frames	356
	Changing the Current View Frame	357
Chapter 18	Customizing the Look and Feel	358
	Understanding the Architecture	358
	IlvLookAndFeelHandler	359
	IlvObjectLFHandler	359
	Class Diagram	360
	Making a User-Defined Component Look-and-Feel Dependant	361
	Creating a New Component	362
	Defining the Object Look-and-Feel Handler API	362
	Subclassing the Object Look-and-Feel Handler	363

Installing the Object Look-and-Feel Handlers	364
Changing the Look and Feel of an Existing Component	364
Subclassing the Component Object Look-and-Feel Handler	364
Replacing an Object Look-and-Feel Handler	365
Creating a New Look-and-Feel Handler	366
Registering a New Look-and-Feel Handler	366
Registering Object Look-and-Feel Handlers Into a New Look-and-Feel Handler	367

Part III IBM ILOG Views Application Framework..... 368

Chapter 19 Introducing IBM ILOG Views Application Framework.....	370
What is Application Framework	370
The Document/View Architecture.....	371
Chapter 20 Using the Application Framework Editor	374
Starting Up the Application Framework Editor	374
Application Framework Editor Main Window	375
Components Palette	376
Workspace	377
Creating a New Application	378
Selecting a Document Type	379
Creating and Configuring an Options File (.odv file)	380
Setting Application Parameters	380
Adding Menu Items	381
Adding Toolbar Items	382
Setting Document Parameters	382
Setting General Document Parameters	383
Setting Parameters for a Selected Document	385
Setting Window Parameters	386
Setting Toolbar Parameters for a Document Type	389
Setting Action Parameters	389
Action Definition	390

	Creating an Action	393
	Setting Popup Menu Parameters	393
	Popup Definition.	394
	Creating a Popup Menu	395
	Adding a Popup Item	396
	Adding a New Popup Submenu	396
	Setting Dialog Parameters	396
	Dialog Definition.	397
	Creating a Dialog Box	399
	Setting Data Parameters	400
	Data Definition	401
	Generating Parameters	401
	Parameters Command	401
	GUI Action Summary	405
Chapter 21	Implementing an Application.	408
	How Application Framework Functions.	408
	Option Files	410
	Main File	411
	Implementation of a Document Class	411
	New Document.	412
	Serialization	412
	Commands	414
	Undo / Redo / Repeat Actions	415
	Reflecting Changes Made In the Data to Associated Views	416
	Implementation of a Document View Class.	417
	Interactions.	418
Chapter 22	Application Framework Interfaces	422
	The Interface Mechanism	422
	Declaring an Interface	423
	Naming Convention for Macros	423

Chapter 23 **Actions** **426**

Activating an Action Event426

Processing an Action Event427

Appendix A **Editing States** **428**

Creating a Simple Application428

 Creating the First Panel429

 Creating the Second Panel430

 States Panels432

Editing the Show State433

 Chaining States435

 Changing the Label and the Callback of the Show Button.....437

 Creating a Substate: the Edit State438

 The State File.....442

Index**444**

About This Manual

This *User's Manual* explains how to use IBM® ILOG® Views Controls. It explains three of the packages that make up IBM ILOG Views Controls: IBM ILOG Views Studio, IBM ILOG Views Gadgets, and IBM ILOG Views Application Framework.

What You Need to Know

This manual assumes that you are familiar with the PC or UNIX environment in which you are going to use IBM® ILOG® Views, including its particular windowing system. Since IBM ILOG Views is written for C++ developers, the documentation also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Manual Organization

This manual contains three separate parts divided into chapters, and one Appendix. Each of these parts describes one of the packages that make up IBM® ILOG® Views Controls.

- ◆ **Part I, *Creating GUI Applications with IBM ILOG Views Studio*** describes how to use IBM ILOG Views Studio with the Gadgets extension installed.

- ◆ **Part II, *IBM ILOG Views Gadgets*** provides information for developing applications that incorporate IBM ILOG Views Gadgets.

- ◆ **Part III, *IBM ILOG Views Application Framework*** describes how to use the Application Framework package of IBM ILOG Views.

Appendix A, *Editing States* provides an example of how to use the state mechanism of IBM ILOG Views Studio.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- ◆ Code extracts and file names are written in `courier` typeface.
- ◆ Entries to be made by the user are written in `courier`.
- ◆ Some words appear in *italics* when seen for the first time.

Naming Conventions

Throughout this manual, the following naming conventions apply to the API.

- ◆ The names of types, classes, functions, and macros defined in the library begin with `Ilv`.
- ◆ The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

```
class IlvDrawingView;
```

- ◆ The names of virtual and regular methods begin with a lowercase letter; the names of static methods start with an uppercase letter. For example:

```
virtual IlvClassInfo* getClassInfo() const;  
static IlvClassInfo* ClassInfo*() const;
```

Part I

Creating GUI Applications with IBM ILOG Views Studio

This part describes how to use IBM® ILOG® Views Studio with the Gadgets extension installed. It contains the following chapters:

- ◆ Chapter 1, *Introducing the Gadgets Extension of IBM ILOG Views Studio* provides basic information for using the Gadgets extension of Studio. It explains how to launch the GUI applications plug-in, describes the additional interface items that differ from the Foundation Studio, contains a list of additional Studio commands, and describes the interface panels of the Gadgets extension.
- ◆ Chapter 2, *Editing Gadget Panels* tells you how to create a panel by adding gadgets to it and how to edit these objects using the various tools and commands available in IBM ILOG Views Studio.
- ◆ Chapter 3, *Editing Applications* tells you how to create applications using IBM ILOG Views Studio.
- ◆ Chapter 4, *Using the Generated Code* explains how to create a miniature application made up of two panels and how to generate the C++ code for that application.

- ◆ Chapter 5, *Customizing the Gadgets Extension of IBM ILOG Views Studio* provides a list of configuration options that you can use to customize IBM ILOG Views Studio when the Gadgets extension is installed.
- ◆ Chapter 6, *Extending IBM ILOG Views Studio* tells you how to adapt IBM ILOG Views Studio to your specific requirements.
- ◆ Chapter 7, *Using Inspector Classes* tells how to define a new inspector panel and incorporate it into IBM ILOG Views Studio.

At the end of this manual, you will also find the following appendix:

- ◆ Appendix A, *Editing States* shows how to use IBM ILOG Views Studio to set up different states for application panels.

Introducing the Gadgets Extension of IBM ILOG Views Studio

This chapter introduces you to the Gadgets extension of IBM® ILOG® Views Studio. You can find information on the following topics:

- ◆ *Loading the GUI Application and GUI Generation Plug-In*
- ◆ *The Main Window*
- ◆ *The Palettes Panel*
- ◆ *Gadgets Extension Commands*
- ◆ *Gadgets Extension Panels*

Note: *The chapters concerning the use of the Gadgets extension of IBM ILOG Views Studio assume that you are familiar with the information in the IBM ILOG Views Studio User's Manual.*

Loading the GUI Application and GUI Generation Plug-In

Once the Gadgets package of IBM® ILOG® Views has been installed, you can use the GUI Application plug-in and the GUI Generation plug-in with IBM® ILOG® Views Studio.

Launch `ivfstudio` with the `-selectPlugIns` command line parameter. When the IBM ILOG Views Studio Plug-Ins dialog box appears, select the GUI Application (`smguiapp`) and the GUI Generation (`smguigen`) check box and click OK.

You can also execute the `SelectPlugins` command from the Studio Main window to display the IBM ILOG Views Studio Plug-Ins dialog box. Then select the GUI Application (`smguiapp`) and the GUI Generation (`smguigen`) check box and click OK.

The Main Window

When you launch the application, the Main window of IBM® ILOG® Views Studio appears as follows:

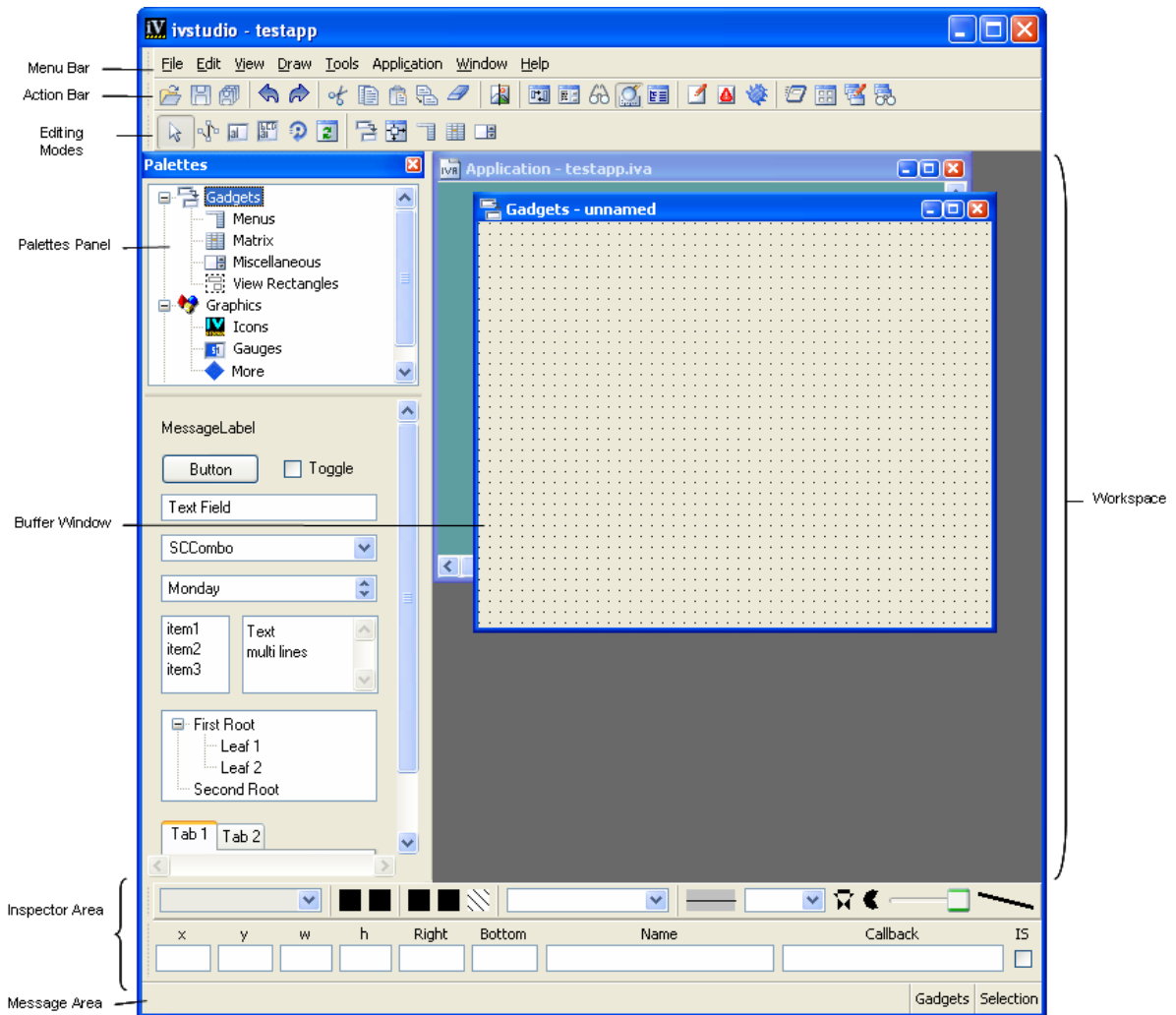


Figure 1.1 IBM ILOG Views Studio Main Window with Gadgets Extension at Start-up Time

The Main window appears much as it does when only the Foundations package is installed. However, you will notice that with Gadgets package you have access to additional buffer windows, additional palettes in the Palettes panel, and additional items in the menu bar and toolbars of the interface.

Buffer Windows

Applications and panels are created in the buffer windows displayed in the Main window. The current buffer type is shown at the bottom of the Main window.

With the Gadgets extension of IBM® ILOG® Views Studio, you can edit the following types of buffers:

- ◆ Gadgets
- ◆ 2D Graphics
- ◆ Application

An empty Gadgets buffer and an empty Application buffer are displayed by default when you launch IBM ILOG Views Studio.

When creating a new buffer window, you must specify its type (Gadgets, 2D Graphics, or Application) using the File > New menu selection.

You will notice the following differences as you switch between the buffers currently loaded in the Main window:

- ◆ Each buffer type has its own set of editing modes. When you change the current buffer, the editing modes available as icons in the toolbar change accordingly.
- ◆ The behavior of certain commands varies depending on the current buffer. For example, the Test command tests just the panel if you are editing a Gadgets buffer, but it tests all the panel instances in an application if you are editing an Application buffer.

The Gadgets Buffer Window

The Gadgets buffer window is used to edit panel classes. It lets you edit the contents of an `IlvGadgetContainer` object. Gadgets can be dragged from the Palettes panel to the active Gadgets buffer window.

To open a new Gadgets buffer window:

1. Choose New from the File menu.
2. Then choose Gadgets from the submenu that appears.

When you open a `.ilv` file that was generated by an `IlvGadgetContainer`, a Gadgets buffer window is automatically opened.

For more information on editing gadgets buffer windows, see Chapter 2, *Editing Gadget Panels*.

The 2D Graphics Buffer Window

The 2D Graphics buffer is the default for the Foundations package. It is still available with the Gadgets extension of IBM ILOG Views Studio. It allows you to edit the contents of an `IlvManager` or an `IlvContainer`. It uses an `IlvManager` to load, edit, and save objects.

To create a new 2D Graphics buffer window:

1. Choose New from the File menu.
2. Then choose 2D Graphics from the submenu that appears.

To open this window, you can also execute the `NewGraphicBuffer` command from the Commands panel, which you can display by choosing `Commands` from the `Tools` menu.

When you open a `.ilv` file that was generated by an `IlvManager`, a 2D Graphics buffer window is automatically opened.

The Application Buffer Window

The Application buffer lets you edit the contents and properties of an `IlvApplication`.

To create a new Application buffer window:

1. Choose `New` from the `File` menu.
2. Then choose `Application` from the submenu that appears.

You can also execute the `NewApplication` command from the Commands panel, which you can display by choosing `Commands` from the `Tools` menu.

When you open a `.iva` file, the Application buffer discards its contents and edits the newly opened application.

Editing an Application

In IBM ILOG Views Studio, an application can be edited in the same way as other types of buffers. Only one application may be open at a time. Opening a new application automatically closes any open application.

When you launch IBM ILOG Views Studio, a default Application buffer window, called `testapp`, is opened.

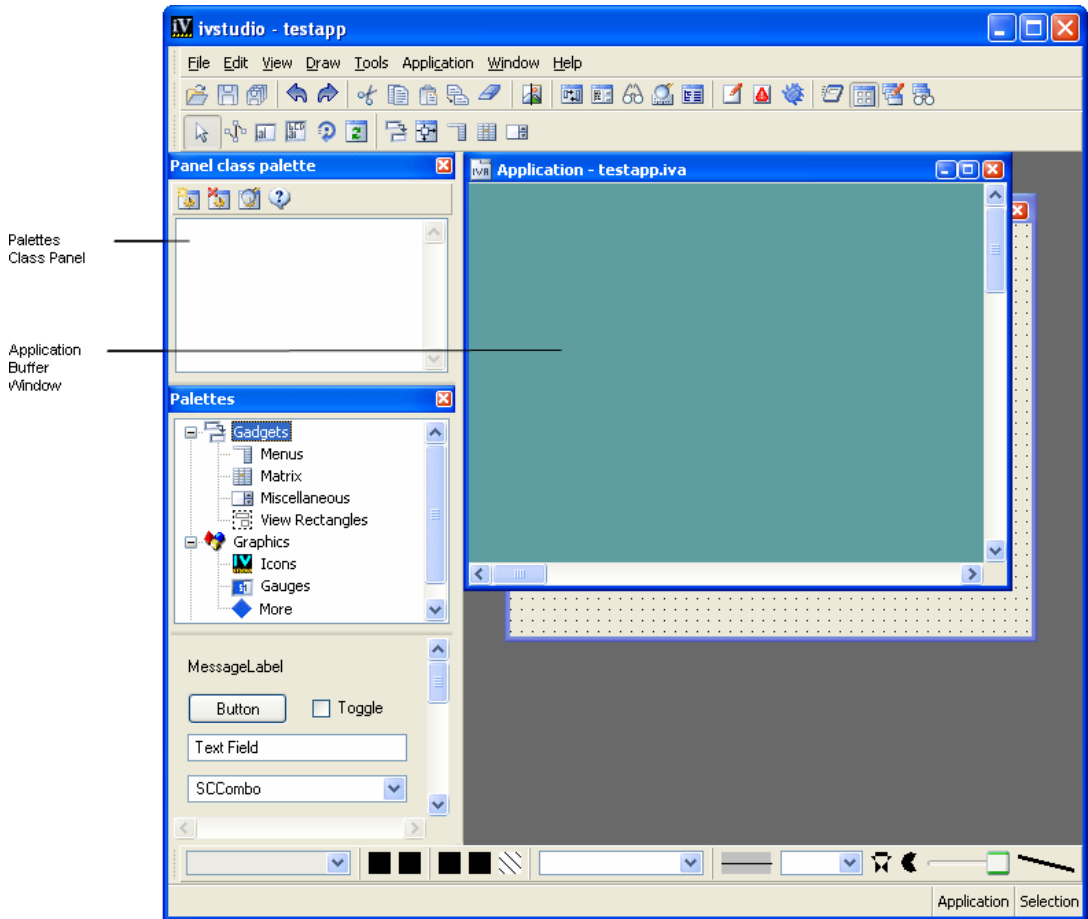


Figure 1.2 *The Application Buffer Window*

You edit an application by adding panel class instances via the Panel Class palette. The Panel Class palette is a palette that you use to create, inspect, or remove panel classes. The panel classes that are available in the Panel Class palette may be dragged directly into the Application buffer window to create panel instances.

For more information on how to edit applications, see Chapter 3, *Editing Applications*.

The Menu Bar

When the Gadgets package is installed, additional commands are available through the menu bar in the Main window. Most notably, you will notice the addition of the Application menu, which provides access to the commands for generating the C++ code of your application.

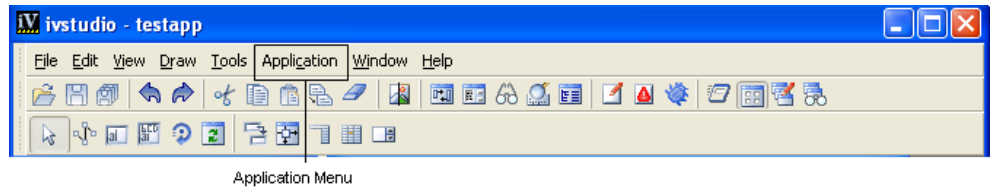


Figure 1.3 IBM ILOG Views Studio Gadgets Extension Menu Bar

The following tables summarize the additional commands that you can execute through the menu bar. For details on these commands, see *Gadgets Extension Commands* on page 39, where they are listed in alphabetical order.

File Menu Commands

Menu Item	Command
New > Gadgets	<i>NewGadgetBuffer</i>
New > Application	<i>NewApplication</i>
New > Make Default Application	<i>MakeDefaultApplication</i>

Application Menu Commands

Menu Item	Command
Test Panel	<i>TestPanel</i>
New Panel Class	<i>NewPanelClass</i>
Panel Class Palette	<i>ShowClassPalette</i>
Panel Class Inspector	<i>ShowPanelClassInspector</i>
Generate Panel Subclass	<i>GeneratePanelSubClass</i>
Add Panel Instance	<i>AddPanel</i>
Panel Inspector	<i>InspectPanel</i>
Application Inspector	<i>ShowApplicationInspector</i>
Test Application	<i>TestApplication</i>
Generate	<i>Generate</i>

Menu Item	Command
Generate All	<i>GenerateAll</i>
Generate Application	<i>GenerateApplication</i>
Generate Panel Class	<i>GeneratePanelClass</i>
Generate Make File	<i>GenerateMakeFile</i>

Window Menu Commands

Menu Item	Command
Edit Application	<i>EditApplication</i>

The Action Toolbar

The Action toolbar contains additional icons for you to quickly access the commands of the Gadgets extension.

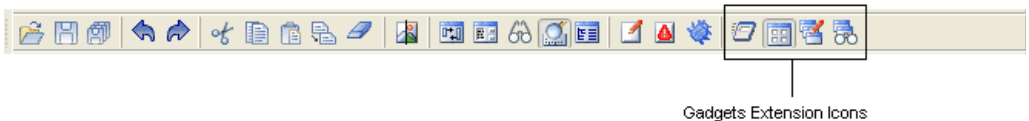


Figure 1.4 IBM ILOG Views Studio Gadgets Extension Action Toolbar



Test Tests the application if the current buffer is the application or tests the panel data if the current buffer is a panel buffer. See *TestDocument* on page 48.



Panel Class Palette Shows or hides the Panel Class Palette in the Main window. See *ShowClassPalette* on page 48.



Edit Application Make the application buffer the active buffer and shows the Panel Class Palette. See *EditApplication* on page 40.



Application Inspector Opens the Application Inspector panel. See *ShowApplicationInspector* on page 47.

The Editing Modes Toolbar

The editing modes available for your use depend on the type of buffer you are editing. You will see different icons in displayed in the toolbar for each of the buffers available with the Gadgets extension.

Gadgets Buffer Editing Modes

The Editing Modes toolbar appears as follows when the Gadgets buffer is the active window in the workspace.

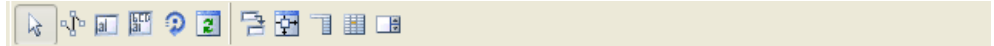


Figure 1.5 Gadgets Buffer Editing Modes Toolbar

The Editing Modes toolbar of the Gadgets buffer contains the following icons:



Selection Mode Use the Selection mode for selecting, creating, deleting, moving, resizing and performing other common editing operations. This mode is selected when IBM ILOG Views Studio is launched.



PolySelection Mode Use this mode to move or rotate the different points of your `IlvPolyline`, `IlvPolygon`, `IlvSpline`, `IlvFilledSpline`, and `IlvClosedSpline` objects. To complete the operation, double-click the workspace or select another mode.



Label Mode Use this mode to create and edit an `IlvLabel` object. After selecting this interactor, click the workspace to indicate the label position and type the string you want. Press Enter to complete the operation.

To edit an existing `IlvLabel` object, select this mode and click the `IlvLabel` object you want to edit.



Label List Mode Use this mode to create and edit a multiline label (`IlvListLabel`) object. After selecting this interactor, click the workspace to indicate the label position and type the string you want. You can go to a new line by pressing Enter. Double-click the workspace (outside this object) to complete the operation.

To edit an existing `IlvListLabel`, select this mode and click the `IlvListLabel` you want to edit.



Rotate Mode Use this mode to rotate an object in the buffer window. First, select the object you want to rotate in the buffer window. Click the Rotate Mode icon in the Editing Modes toolbar. Then click the left mouse button in the buffer window. An arrow appears in the buffer window. Drag the mouse to indicate the angle of rotation. When you release the mouse button, the object will rotate the specified amount.



Active Mode Use the Active mode to test the behavior of your objects and edit some of their properties. In the Active mode, the objects in the workspace can respond to mouse and keyboard events. You can thus change text field labels, toggle the state of a toggle button, and so on.



Focus Mode Use this mode to specify the path of the keyboard focus in your panel. See *Setting the Keyboard Focus in Panels* on page 60.



Attachments Mode Use this mode to set how the position and dimensions of the objects in the panel change when the panel is resized. See *Using the Attachments Mode* on page 61.



Menu Mode Use this mode to attach pop-up menus to menu bars or other pop-up menus. See *Attaching Pop-up Menus to the Menu Bar* on page 71.



Matrix Mode Use this mode to change the matrix items that appear in a matrix gadget. See *Using Matrices* on page 75.



Spin Box Mode Use this mode to specify the items that appear in a spin box object. See *Editing Spin Boxes* on page 78.

Application Buffer Editing Modes

The Editing Modes toolbar appears as follows when the Application buffer is the active window in the workspace.



Figure 1.6 Application Buffer Editing Modes Toolbar

The Editing Modes toolbar of the Gadgets buffer contains the following icons:



Generate Use this mode to generate the application and modified panels.

Graphics Buffer Editing Modes

When you use a Graphics buffer, you have access to the same editing modes that you use with the Foundation Studio.



Figure 1.7 Graphics Buffer Editing Modes Toolbar

These editing modes are described in Chapter 3, “*The IBM ILOG Views Studio Interface*,” of the *IBM ILOG Views Studio User’s Manual*.

The Palettes Panel

When you use the Gadgets extension of IBM® ILOG® Views Studio, you have access to additional predefined gadget objects through the Palettes panel. You will use the gadgets in

the Palettes panel just in the same as you do with the Foundations package of IBM ILOG Views Studio. You create the objects in the workspace either using the drag-and-drop operation or the creation mode operation.

You will notice the 5 additional palettes in the upper pane of the Palettes panel that are provided with the Gadgets extension. Click the appropriate palette in the upper pane to access the various gadget objects in the lower pane.

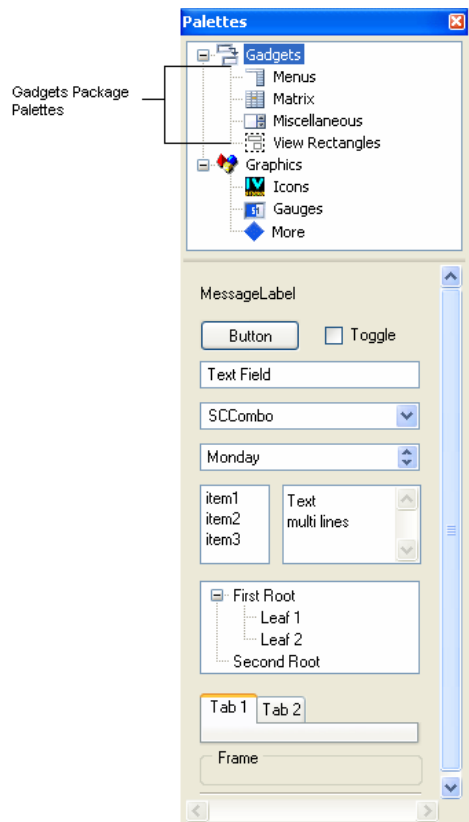


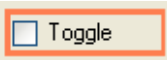


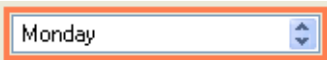


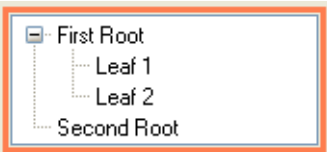


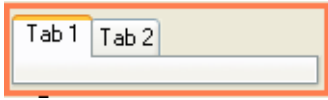
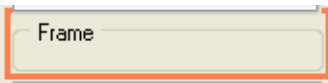

Figure 1.8 IBM ILOG Views Studio Gadgets Extension Palettes Panel

The following sections describe the objects provided with the Gadgets extension. For a description of the objects provided with the Foundation package, see the *IBM ILOG Views Studio User's Manual*.

Gadgets Palette



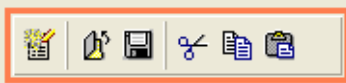
The Gadgets palette contains the following objects that can be created using the normal creation mode of IBM® ILOG® Views Studio or the drag-and-drop operation.

Type	Icon	Description
Message Label		Creates an <code>IlvMessageLabel</code> object.
Button		Creates an <code>IlvButton</code> object.
Toggle		Creates an <code>IlvToggle</code> object.
Text Field		Creates an <code>IlvTextField</code> object.
SC Combo Box		Creates an <code>IlvScrolledComboBox</code> object.
Spin Box		Creates an <code>IlvSpinBox</code> object.
String List		Creates an <code>IlvStringList</code> object.
Multiline Text		Creates an <code>IlvText</code> object.
Tree		Creates an <code>IlvTreeGadget</code> object.

Type	Icon	Description
Notebook		Creates an <code>IlvNotebook</code> object.
Frame		Creates an <code>IlvFrame</code> object.
Relief Line		Creates an <code>IlvReliefLine</code> object.

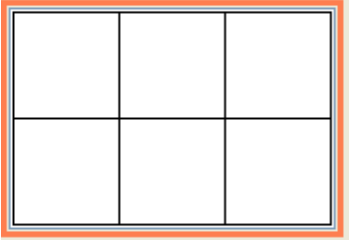

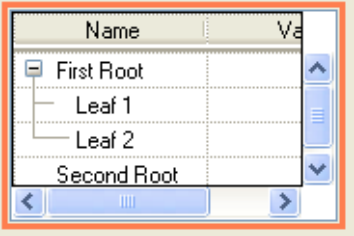
Menus Palette

The Menus palette contains the following objects that can be created using the normal creation mode of IBM® ILOG® Views Studio or the drag-and-drop operation.

Type	Icon	Description
Menu Bar		Creates an <code>IlvMenuBar</code> object.
Pop-up Menu		Creates an <code>IlvPopupMenu</code> object. If you want to attach a pop-up menu to a menu bar or another pop-up menu, you must use the Menu editing mode. Click the Menu icon in the Editing Modes toolbar.
Toolbar		Creates an <code>IlvToolBar</code> object.



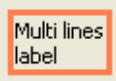





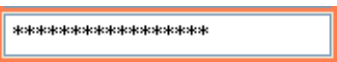

Matrix Palette



The Matrix palette contains the following objects that can be created using the normal creation mode of IBM® ILOG® Views Studio or the drag-and-drop operation.

Type	Icon	Description
Matrix		Creates an <code>IlvMatrix</code> object.
Sheet		Creates an <code>IlvSheet</code> object.
Hierarchical Sheet		Creates an <code>IlvHierarchicalSheet</code> object.

Miscellaneous Palette


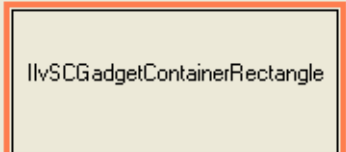
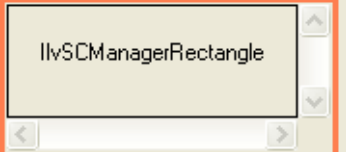
The Miscellaneous palette contains the following objects that can be created using the normal creation mode of IBM® ILOG® Views Studio or the drag-and-drop operation.

Type	Icon	Description
Slider		Creates an <code>IlvSlider</code> object. You can choose either a horizontal or a vertical orientation for the slider.
Label		Creates an <code>IlvLabel</code> object.
Multilines Label		Creates an <code>IlvListLabel</code> object.
Scroll bar		Creates an <code>IlvScrollBar</code> object. You can choose either a horizontal or a vertical orientation for the scroll bar.
Option menu		Creates an <code>IlvOptionMenu</code> object.
Combo Box		Creates an <code>IlvComboBox</code> object.
Number Field		Creates an <code>IlvNumberField</code> object.
Date Field		Creates an <code>IlvDateField</code> object.
Password Field		Creates an <code>IlvPasswordTextField</code> object.
Colored Toggle		Creates an <code>IlvColoredToggle</code> object.

Type	Icon	Description
Arrow Button		Creates an <code>IlvArrowButton</code> object.
Gadget		Creates an <code>IlvGadget</code> object.

View Rectangles Palette

The View Rectangles palette contains the following objects that can be created using the normal creation mode of IBM® ILOG® Views Studio or the drag-and-drop operation.

Type	Icon	Description
Gadget Container		Creates an <code>IlvGadgetContainerRectangle</code> object.
SC Gadget Container		Creates an <code>IlvSCGadgetContainerRectangle</code> object.
SC Manager		Creates an <code>IlvSCManagerRectangle</code> object.

Gadgets Extension Commands

This section presents an alphabetical listing of the additional predefined commands that are available in the Gadgets extension of IBM® ILOG® Views Studio. (All of the

IBM ILOG Views Studio Foundation commands are available as well.) For each command, it indicates its label, the category to which it belongs, how to access it if it is accessible other than through the Commands panel, and what it is used for.

To display the Commands panel, choose Commands from the Tools menu in the Main window or click the Commands icon  in the Action toolbar.

AddPanel

Label	Add Panel
Category	application
Action	Creates a panel instance of the selected panel class and adds it to the application.

EditApplication

Label	Edit Application
Path	Main window: Tools menu and Edit application icon in the toolbar
Category	application
Action	Selects the application buffer and opens the Class Palette.

Generate

Label	Generate
Path	Main window: Application menu
Category	application
Action	Saves the application description file, and generates the C++ code for the application and its modified panel classes. If the application file name is the default name, a File Selector panel is opened to let you enter a new file name. If the Make toggle button in the Application Inspector is turned on, the make file is also generated.

GenerateAll

Label	Generate All
Path	Main window: Application menu
Category	application
Action	Saves the application description file, and generates the C++ code for the application and all its panel classes. If the application file name is the default name, a File Selector panel is opened to let you enter a new file name. If the Make toggle button in the Application Files inspector is turned on, the make file is generated too.

GenerateApplication

Label	Generate Application
Path	Main window: Application menu
Category	application
Action	Saves the application description file, and generates the C++ code for the application. If the application file name is the default name, a File Selector panel is opened to let you enter a new file name. If the Make toggle button in the Application Inspector is turned on, the make file is also generated.

GenerateMakeFile

Label	Generate Make File
Path	Main window: Application menu
Category	application
Action	Generates the application make file, even if the Make toggle button in the Application Inspector is not turned on.

GeneratePanelClass

Label	Generate Panel Class
Path	Main window: Application menu
Category	application
Action	Generates the C++ code for the currently selected panel class. To select a panel class, use the Classes palette.

GeneratePanelSubClass

Label	Generate Panel Sub Class
Path	Main window: Application menu
Category	application
Action	Generates a subclass skeleton for the current panel class. This command activates a dialog box which lets you enter the class name, the file base name and the directories where the header and source files will be generated.

InspectPanel

Label	Inspect Panel Class
Path	Panel Class palette: toolbar
Category	application, panel
Action	Opens the Panel Class Inspector that lets you inspect the properties of the selected panel class.

KillTestPanels

Label	Kill Test Panels
--------------	------------------

Category	application
Action	Kills all the panels that are created for testing the application or the current buffer.

MakeDefaultApplication

Label	Make Default Application
Path	Main window: File menu > New
Category	application
Action	If the current buffer is a panel buffer, this command creates an application, creates a panel class for the current buffer, and creates a panel instance of that panel class.

NewApplication

Label	Application
Path	Main window: File menu > New
Category	application
Action	Discards the current application and edits a new one. Only one application can be edited at a time.

NewGadgetBuffer

Label	Gadgets
Path	Main window: File menu > New
Category	buffer
Action	Creates a new gadget buffer. This buffer becomes the current buffer.

NewPanelClass

Label	New Panel Class
Path	Panel Class palette
Category	application
Action	Creates a new panel class for the current buffer if this buffer is designed for a container and if it is not already part of the application.

OpenApplication

Label	Open...
Path	Main window: File menu and Open icon in the toolbar
Category	application
Action	Discards the current application and loads an application previously saved by IBM ILOG Views Studio. This command opens a File Selector panel that lets you choose an application description file.

RemoveAllAttachments

Label	Remove All Attachments
Category	attachments
Action	Removes all the attachments for the selected object. This command only works if the current mode is the Attachments mode.

RemoveAttachments

Label	Remove Attachments
Category	attachments
Action	Removes all the attachments of the current buffer. This command only works if the current mode is the Attachments mode.

RemovePanel

Label	Remove Panel
Path	Panel instance window in the Application buffer window: the close (X) button or the menu.
Category	application
Action	Removes the selected panel instance from the application.

RemovePanelClass

Label	Remove Panel Class
Path	Panel Class palette
Category	application
Action	Removes the selected panel class and all its panel instances.

SaveApplication

Label	Save
Path	Main window: File menu and Save icon in the toolbar
Category	application
Action	Saves the description of the current application to its data file. If the application's name is the default name, this command executes the <code>SaveApplicationAs</code> command in order to let you enter an application file name. Execute the <code>SaveApplicationAs</code> command for saving a new application for the first time or changing its file name.

SaveApplicationAs

Label	Save As...
Path	Main window: File menu and Save icon in the toolbar

Category	application
Action	Opens a File Selector panel that lets you enter a new file name for your application and save its description to that file. This command lets you change your application's file base name (therefore its generated C++ files) as well as its location.

SelectAttachmentsMode

Label	Attachments
Path	Main window: Editing Modes toolbar (when editing Gadgets buffers)
Category	mode, gadgets
State	True if this mode is selected.
Action	Selects the Attachments mode. See <i>Using the Attachments Mode</i> on page 61.

SelectFocusMode

Label	Focus
Path	Main window: Editing Modes toolbar (when editing Gadgets buffers)
Category	mode, gadgets
State	True if this mode is selected.
Action	Selects the Focus mode. See <i>Setting the Keyboard Focus in Panels</i> on page 60.

SelectMatrixMode

Label	Matrix
Path	Main window: Editing Modes toolbar (when editing Gadgets buffers)
Category	mode

State	True if this mode is selected.
Action	Selects the Matrix mode. See <i>Using Matrices</i> on page 75.

SelectMenuMode

Label	Menu
Path	Main window: Editing Modes toolbar (when editing Gadgets buffers)
Category	mode
State	True if this mode is selected.
Action	Selects the Menu mode. See <i>Editing Menus</i> on page 66.

ShowAllTestPanels

Label	Show All Test Panels
Category	application, panel
Action	Shows all the current panels in the application. Unlike the <code>TestApplication</code> command that shows only visible panels, this command shows every panel.

ShowApplicationInspector

Label	Inspect Application
Path	Main window: Tools menu and Inspect Application in the toolbar
Category	application, panel
Action	Opens the Application Inspector panel.

ShowClassPalette

Label	Classes
Path	Main window: Classes icon in the toolbar
Category	application, panel
Action	Opens the Class Palette that lets you create, inspect or remove panel classes and creates panel instances.

ShowPanelClassInspector

Label	Inspect Panel Class
Path	Panel Class palette: toolbar
Category	application, panel
Action	Shows or hides the Panel Class Inspector of the selected object.

TestApplication

Label	Test
Path	Main window: toolbar
Category	application
State	True if the application is being tested.
Action	If the application is not being tested, opens the panels that are visible and lets you test them until you execute this command again. If the application is being tested, this command kills the test panels and stops testing.

TestDocument

Label	Test Application
Path	Main window: toolbar

Category	application
Action	Tests the application if the current buffer is the application or tests the panel data if the current buffer is a panel buffer.

TestPanel

Label	Test
Path	Main window: toolbar
Category	buffer
Action	Tests the current buffer.

Gadgets Extension Panels

The Gadgets extension provides several additional panels and dialog boxes for your use when creating your own panels and applications.

- ◆ *Application Inspector*
- ◆ *The Panel Class Inspector*
- ◆ *The Panel Instance Inspector*

Application Inspector

The Application inspector is used to edit the settings of the generated application. It lets you specify the location of the C++ files, the class declaration, and several options for the generated code. This panel also lets you insert code in the generated application class files (see *The Header and Source Pages* on page 93).

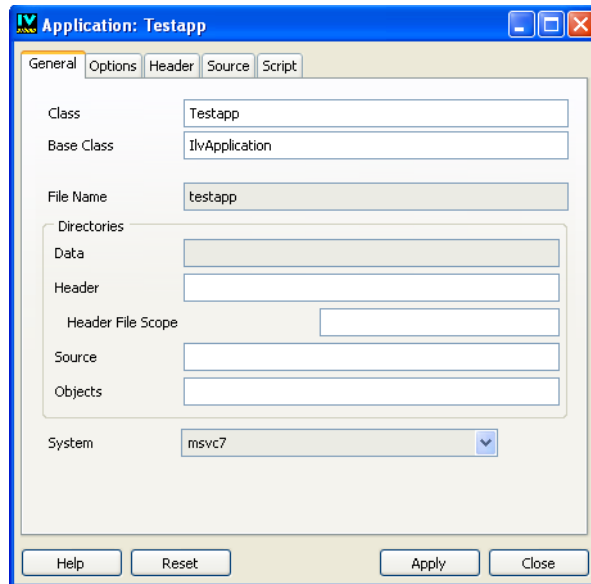
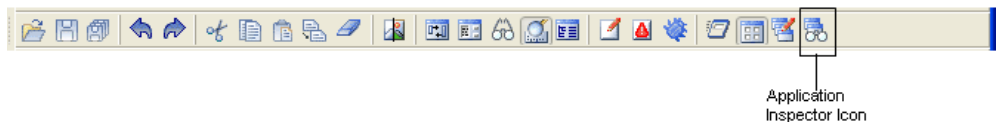


Figure 1.9 Application Inspector

Access to Panel

The panel is accessed by:

- ◆ Clicking the Application Inspector icon in the Action toolbar.



Application Inspector Icon

or

- ◆ Choosing Application Inspector from the Application menu.

or

- ◆ Choosing Commands from the Tools menu, selecting the ShowApplicationInspector command in the list, and clicking Apply.

Application Inspector Elements

The Application inspector has five notebook pages: General, Options, Header, Source, and Script; and four buttons: Apply, Reset, Close, and Help. For a complete description of each notebook page and the fields contained on the page, see *The Application Inspector* on page 89.

The Panel Class Inspector

The Panel Class inspector is used to inspect the directories and options for generating the panel class. This panel can also be used to insert your own code in the generated panel class files (see *The Header and Source Pages* on page 93).

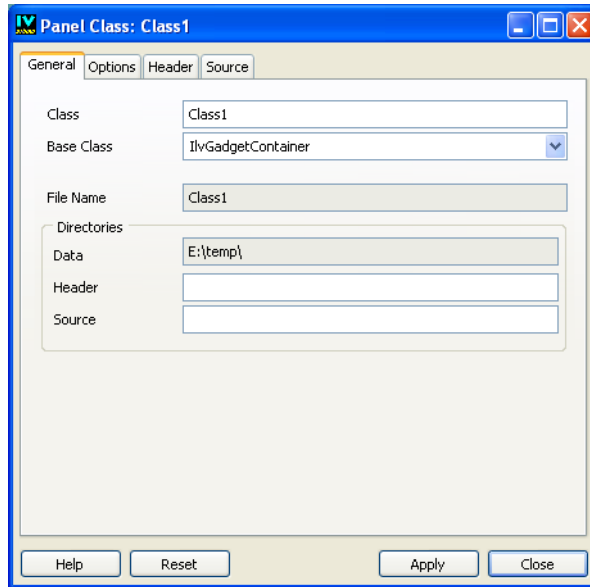
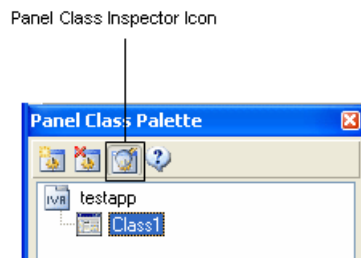


Figure 1.10 Panel Class Inspector

Access to Panel

The panel is accessed by:

- ◆ Clicking the Panel Class Inspector icon in the Panel Class palette.



or

- ◆ Choosing Panel Class Inspector from the Application menu.

or

- ◆ Choosing Commands from the Tools menu, selecting the `ShowPanelClassInspector` command in the list, and clicking Apply.

Panel Class Inspector Elements

The Panel Class Inspector has four notebook pages: General, Options, Header, and Source; and four buttons: Apply, Reset, Close, and Help. For a complete description of each notebook page and the fields contained on the page, see *The Panel Class Inspector* on page 98.

The Panel Instance Inspector

The Panel Instance inspector is used to edit the properties of the selected panel instance.

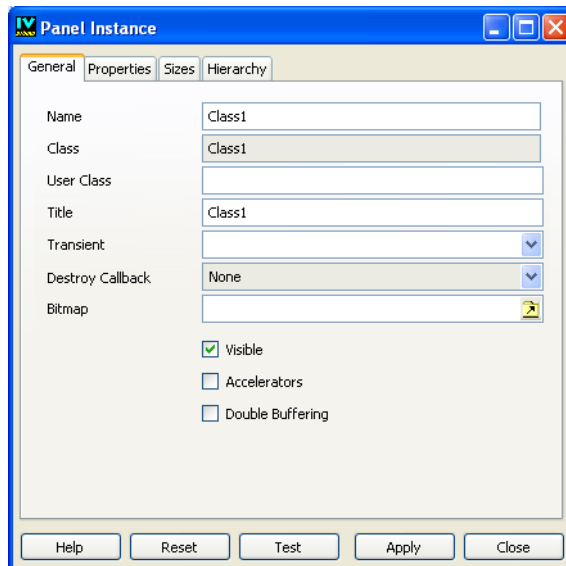


Figure 1.11 Panel Instance Inspector

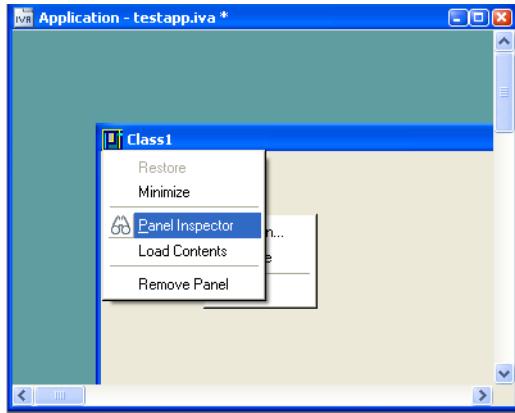
Access to Panel

The panel is accessed by:

- ◆ Double-clicking the title bar of the panel instance in the Application buffer window.

or

- ◆ Choosing Panel Inspector from the panel pop-up menu accessible by clicking the box located in the top-left corner of the panel instance.



or

- ◆ Choosing Panel Inspector from the Application menu.

or

- ◆ Choosing Commands from the Tools menu, selecting the `InspectPanel` command in the list, and clicking Apply.

Panel Class Instance Inspector Elements

The Panel Class Inspector has four notebook pages: General, Properties, Sizes, and Hierarchy; and five buttons: Apply, Reset, Test, Close, and Help. For a complete description of each notebook page and the fields contained on the page, see *Inspecting a Panel Instance* on page 104.

Editing Gadget Panels

This chapter introduces you to the basic commands, panels, and modes that you can use to create gadget panels.

You will find information on the following topics:

- ◆ *Creating a New Panel*
- ◆ *Creating Gadget Objects*
- ◆ *Inspecting an Object*
- ◆ *Testing a Panel*
- ◆ *Using Active Mode*
- ◆ *Setting the Keyboard Focus in Panels*
- ◆ *Using the Attachments Mode*
- ◆ *Editing Menus*
- ◆ *Using Matrices*
- ◆ *Editing Spin Boxes*

Creating a New Panel

When IBM® ILOG® Views Studio is launched, an empty Gadgets buffer window is open, which is ready to be edited. You will create your panel in this Gadgets buffer window. If required, to create a new Gadgets buffer window:

1. Choose New from the File menu.
2. Then choose Gadgets in the submenu that appears.

The new Gadgets buffer window becomes the current window and can be edited.

Creating Gadget Objects

The Gadgets palette in the Palettes panel provides the various predefined gadget objects from which you will create the objects for your panels. You can use either a drag-and-drop operation or the creation mode feature.

Using the Drag-and-Drop Operation

When you use the drag-and-drop operation for creating your objects, the object that is added to the buffer window is an exact copy of the object as it is found in the Palettes panel. The object has the same shape and dimensions of the object in the Palettes panel.

To create an object using the drag-and-drop operation:

1. In the upper pane of the Palettes panel, click the item in the tree corresponding to the type of gadget you want to create.

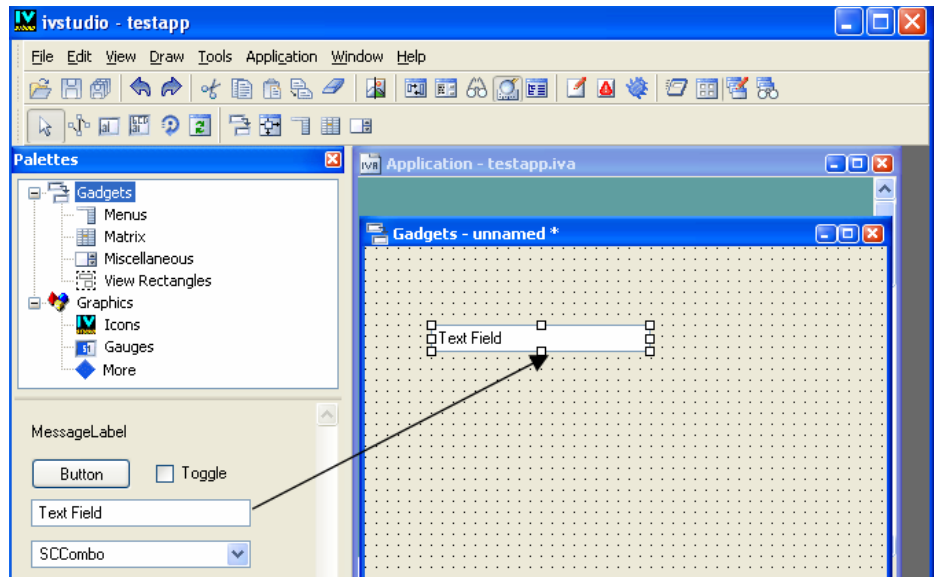
The related palette appears in the lower pane.

2. Click the gadget you are interested in and drag it to the Gadgets buffer window.

When you release the mouse button, you are in Selection mode. The object remains selected in the buffer window and you can modify it as required.

For example, to create a text field:

1. In the upper pane of the Palettes panel, click Gadgets in the tree.
2. In the lower pane of the Palettes panel, click the Text Field gadget.
3. Drag it to the Gadgets buffer window.



Using the Creation Mode

When you use the creation mode, you are essentially drawing the object in the buffer window. You determine for yourself the size and shape of the object. Creation mode also allows you to create multiple objects once you have selected the kind of object you want to create in the Palettes panel.

To create an object using the creation mode:

1. In the upper pane of the Palettes panel, click the item in the tree corresponding to the kind of object you want to create.

The related palette appears in the lower pane.

2. In the lower pane of the Palettes panel, click the object you are interested in. A bounding box appears around the object to indicate that creation mode is active.

If you want to add only one object to the buffer window, click the object in the Palettes window once. (This puts you in transient creation mode. After you have drawn the object in the buffer window, you will leave creation mode automatically.)

If you want to add multiple objects of the same kind, hold down the Shift key and click the object in the Palettes panel. (This puts you in permanent creation mode. You will remain in creation mode and you can draw as many objects as you like. To leave creation mode, you must click the Selection mode icon in the Editing Modes toolbar.)

3. Move the pointer to the buffer window.

4. Click in the buffer window where you want your object positioned and drag the mouse until the object is the size and shape you want.

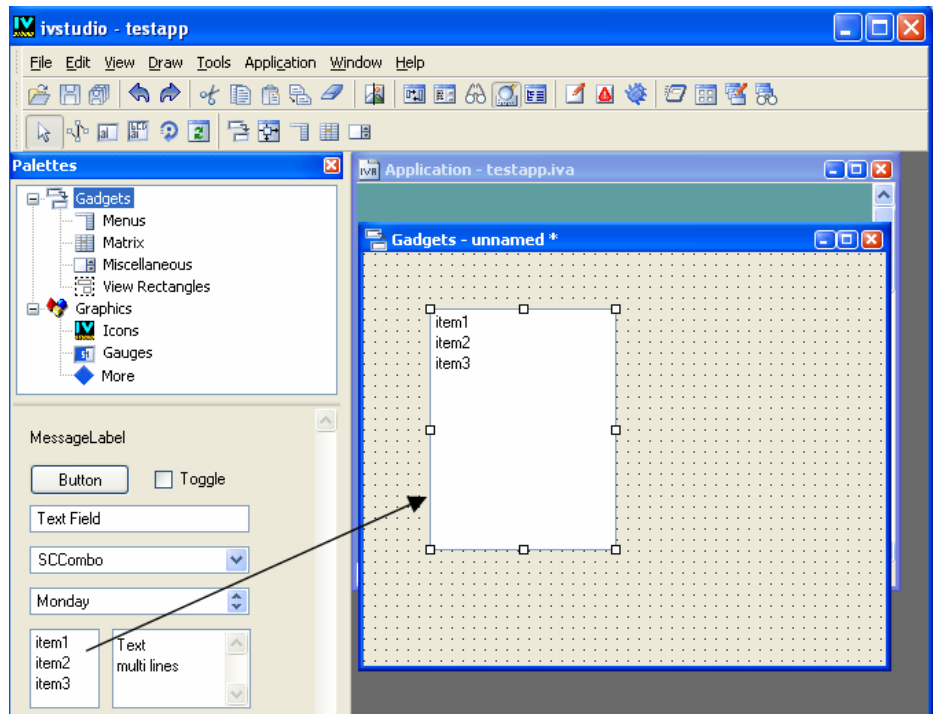
For example, to create a string list box:

1. In the upper pane of Palettes panel, click Gadgets in the tree.
2. In the lower pane of the Palettes panel, click the `IlvStringList` icon once. Notice the bounding box that appears around the `IlvStringList` icon indicating you are in creation mode.


3. Click in the Gadgets window at the position where you want to start drawing the string list box.
4. Drag the mouse until the string list box is the size and shape you want.

As you drag the mouse, you see a bounding box that shows the shape and size of your object.

5. Release the mouse button. The string list box appears with the dimensions of the bounding box you have just drawn.



When you release the mouse button, you automatically leave creation mode and are put into Selection mode. Notice that the `IlvStringList` icon in the Palettes panel is no

longer selected and that the Selection mode icon  is selected in the Editing Modes toolbar.

You can reshape, resize, move, or modify the box as you want.

Inspecting an Object

To inspect the properties specific to an object, double-click the object. You can also click the Inspect icon in the Action toolbar of the Main window.

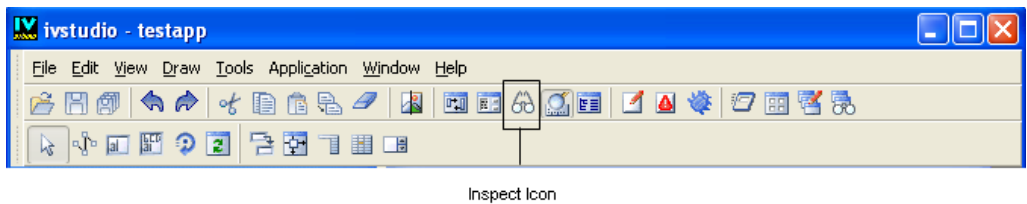


Figure 2.1 Inspect Icon in the Main Window Toolbar

If the object class has an associated inspector panel, you can use it to edit the specific properties of the object class. The contents of the inspector depend on the related object class.

If you click a string list gadget, the following inspector panel appears:

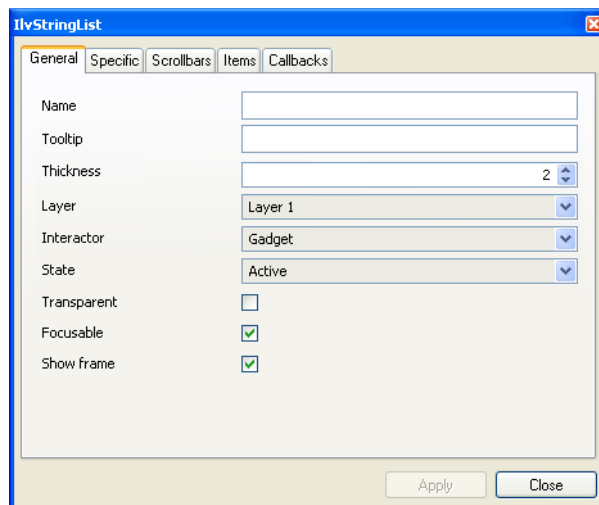


Figure 2.2 String List Inspector Panel (General Page)

To validate the changes made to the object properties, click Apply. To close the panel, click Close.

Only one object can be inspected at a time. If you select another object of the same type while the first is being inspected, the properties of the newly selected object appear in the inspector panel. If another type of object is selected, its associated inspector replaces the one that is displayed.

Testing a Panel

To test the behavior of a panel, click the Test icon in the Main window toolbar.

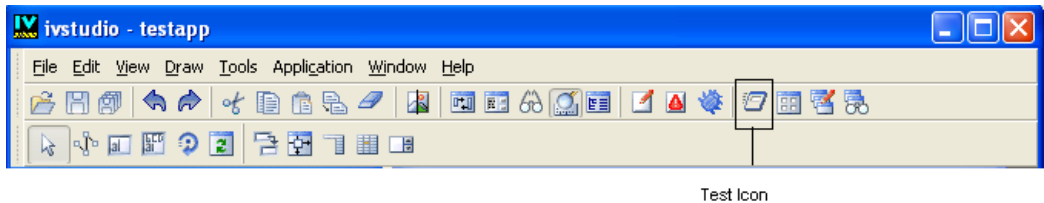


Figure 2.3 Test Icon in the Main Window Toolbar

A panel representing the current buffer is displayed and ready to be tested. To exit the test mode, click the same icon again.

Using Active Mode

In the Active mode, the objects in the workspace can respond to mouse and keyboard events. This lets you test the behavior of your objects and edit some of their properties. You can, for example, change text field labels and toggle the state of a toggle button.

To select the Active mode, click the Active icon in the Editing Modes toolbar:

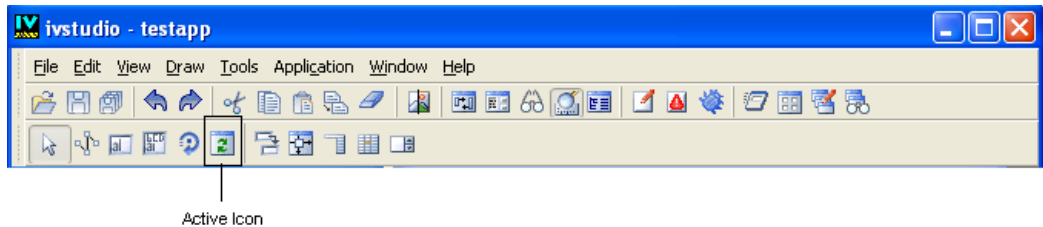
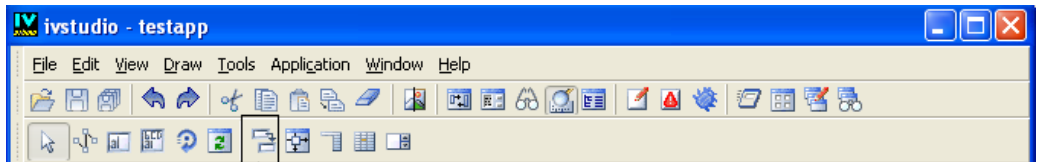


Figure 2.4 Active Mode Icon

Setting the Keyboard Focus in Panels

By default, the keyboard focus is determined by the object bounding box positions. This default focus logically moves between objects from left to right, and from top to bottom. Since this default path is not always suitable, IBM® ILOG® Views Studio provides you with a Focus editing mode that lets you draw the path of the keyboard focus in your panel.

To select the Focus mode, click the Focus icon in the Editing Modes toolbar:



Focus Mode Icon

Figure 2.5 Focus Mode Icon

The keyboard focus path is shown by a series of arrows:

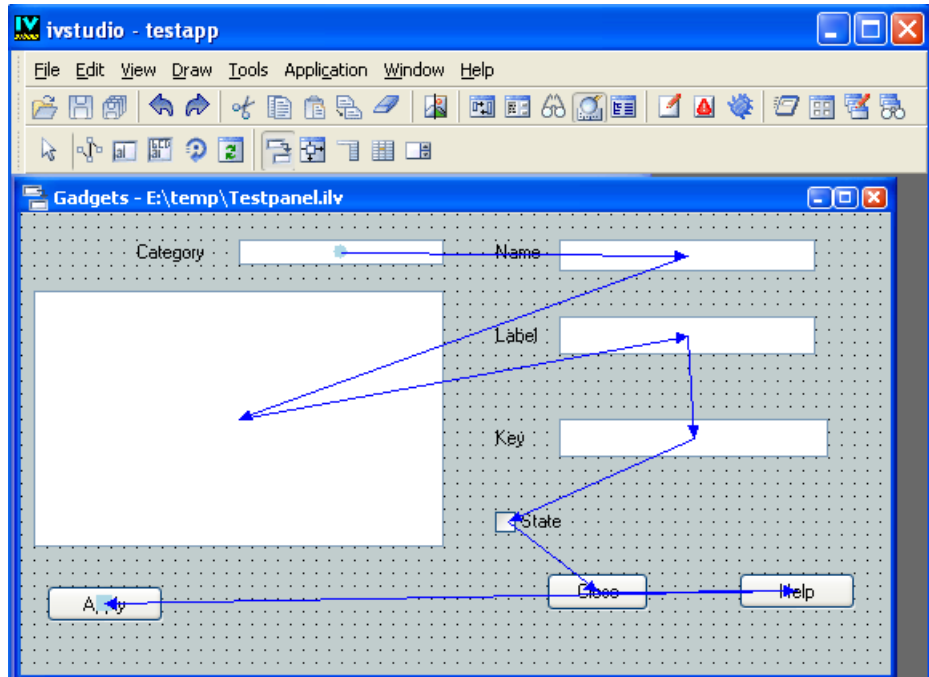


Figure 2.6 Focus Path

Next Focus Object

For each focusable object (graphic object that can receive the keyboard events), you can specify its next focus object. To do so, drag a line from that object and release the mouse button when the line is on the object you want to be the next focus object. This operation only works for focusable objects.

First Focus Object

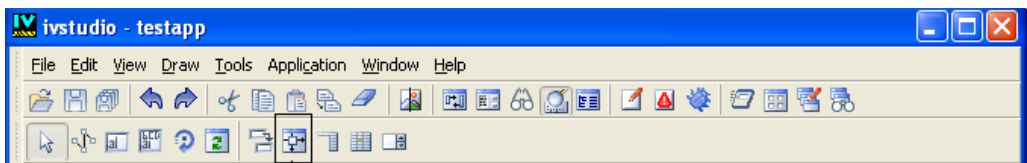
When the panel is focused for the first time, the first focus object is the one that takes the keyboard focus. To designate the first focus object, drag a line from anywhere in the workspace (but not from an object). Release the mouse button when the line is in the object you choose as the first focus object. A filled circle is then drawn in the center of that object. A gadget container can have only one first focus graphic.

Last Focus Object

To designate a last focus graphic, drag a line from that object and release the mouse button when the line is anywhere in the workspace (but not in an object). When the keyboard focus chain leaves a last focus graphic of a gadget container, it goes back to the first focus graphic. However, if the gadget container is linked to another container, it gives the keyboard focus to that container instead. A gadget container can have more than one last focus graphic.

Using the Attachments Mode

IBM® ILOG® Views Studio provides an Attachments mode that you can use to set how the position and dimensions of the objects in the panel change when the panel is resized. To activate this mode, click the Attachments icon in the Editing Modes toolbar:



Attachments Icon

Figure 2.7 Attachments Icon

Setting attachments involves two steps:

- ◆ *Setting the Guides*
- ◆ *Attaching Objects to Guides*

Setting the Guides

When you first click the Attachments icon, guides appear on the top and left borders with numbers next to them. These numbers refer to the weight corresponding to the guides (see below).

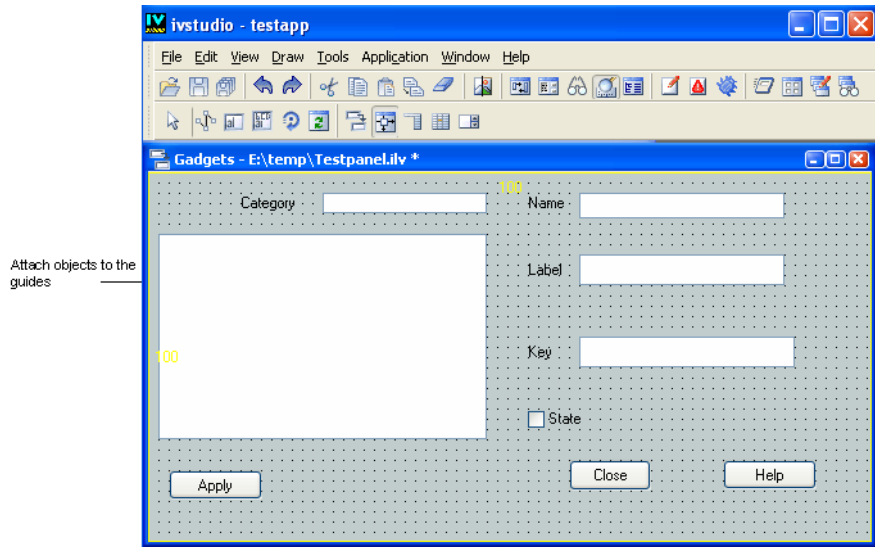


Figure 2.8 Attachment Guides

1. Select a guide by clicking on it or its weight number.
2. Create a guide by selecting one of the initial guides at the top or left borders and dragging the new guide created—with the mouse button pressed—to any position you want, then releasing the mouse button.

A guide is defined by four elements that can be edited in the Guide Inspector panel. To open this panel, double-click the guide or its weight number:

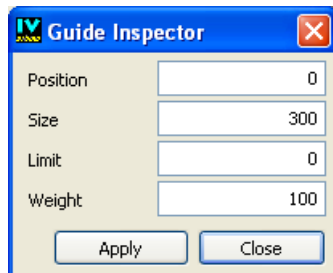


Figure 2.9 Guide Inspector Panel

The Guide Inspector panel contains the following fields:

- ◆ **Position** For horizontal guides, the number of pixels from the top border. For vertical guides, the number of pixels from the left border.
- ◆ **Size** For horizontal guides, the number of pixels to the next guide below. For vertical guides, the number of pixels to the next guide to the right.
- ◆ **Limit** The minimal size of the section set off by the guide when the window is resized (see “Size”).
- ◆ **Weight** The amount of the window to be allocated to a section (delimited by a guide) relative to other sections when the window is resized. The following formula applies to each section when a window is resized, where Delta equals the new size of the window less its initial size:

$$\text{New Section Size} = \text{Initial Section Size} + \text{Delta} \times \frac{\text{Weight of Guide Delimiting the Section}}{\text{Sum of Weights of All Guides}}$$

Attaching Objects to Guides

Each object has an Attachments Inspector that provides added control to set fixed (double lines) and elastic (single line) positions. By double-clicking the object, you open the Attachments Inspector, in which you can only edit existing attachments. The numbers in the text fields refer to the number of pixels from the edge of the object to the guide being used:

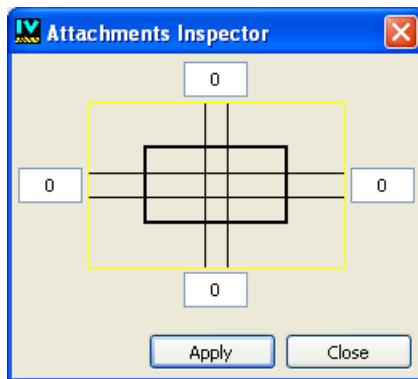


Figure 2.10 Attachments Inspector

There are six locations for creating attachments, each of which can be one of two types:

- ◆ **Location** An attachment can be made from any of the four sides of the object bounding box to a guide parallel to that side. Attachments can also be defined within an object (horizontally and vertically) to specify whether the object changes size when the panel is resized.
 - ◆ **Type** Two types of attachments are possible for each of the six locations: fixed or elastic.
 - **Fixed** (double line): The distance between the object and the guide stays the same as the panel is being resized. Fixed inside the object means that the object does not change size when the panel is resized.
 - **Elastic** (single line): The distance between the object and guide changes proportionately as the panel is being resized. Elastic inside the object means the object changes sizes proportionally when the panel is resized.
-

Attachment Operations

Here are the types of operations you can perform while attaching objects:

- ◆ **Creating Attachments** Select the object you want to attach and drag a line (with the mouse button pressed) from a middle handle of that object to a guide parallel to that side. When the guide becomes highlighted, release the mouse button. Default attachments are made on the opposite side and inside the object. The specified attachments apply to all the selected objects.

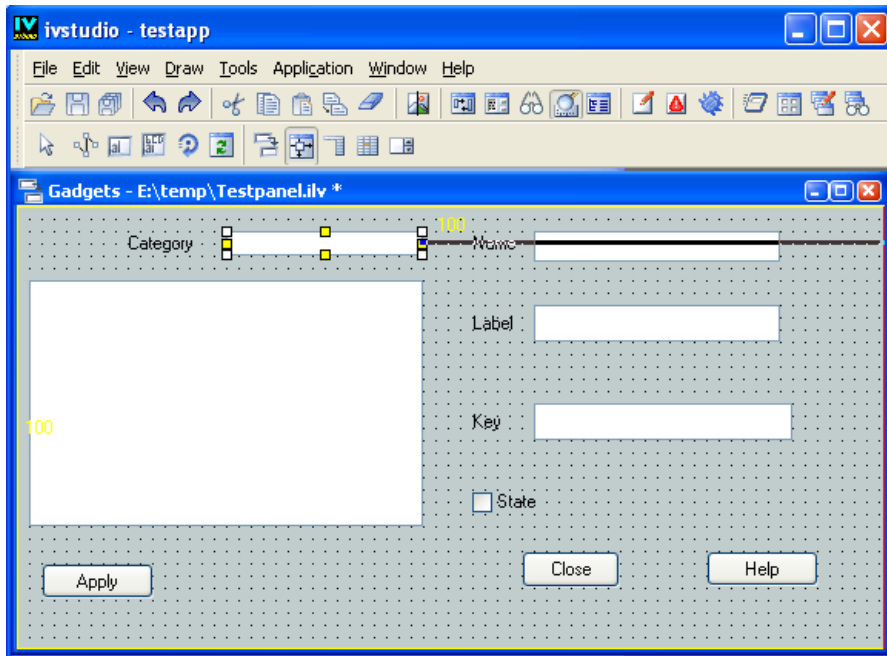


Figure 2.11 Attaching an Object to a Guide

Note: When creating attachments between a guide and an object, you may have problems if the object is too close to the guide. In this case drag the mouse in the opposite direction and then back to the guide.



- ◆ **Removing Attachments** Drag one of the attachment's handles and release the mouse button when the new line is not touching a guide.
- ◆ **Changing an Attachment from One Guide to Another** Drag a line from the attachment handle to a new guide.
- ◆ **Changing the Type of Attachment** Click on the attachment line, which toggles between fixed and elastic. This can also be carried out in the Attachments Inspector.
- ◆ **Showing an Object's Attachments** Select the object in Attachments editing mode and double-click the object to show the Attachments Inspector.

Defaults

The defaults selected depend on the handle used to begin the attachment:

- ◆ **Left and Top Handles** The object enlarges with the panel (fixed, elastic, fixed).
- ◆ **Right and Bottom Handles** The object moves while remaining the same size (elastic, fixed, fixed).

Testing the Attachments

You can test the attachments applied to the objects by clicking the Test icon  in the Main Window toolbar. Change the test window size by using the windowing system. You will see the object behavior as the panel changes size. If you need to make changes, you can click the Test icon  again to close the test panel and make your changes in Attachments mode.

Editing Menu

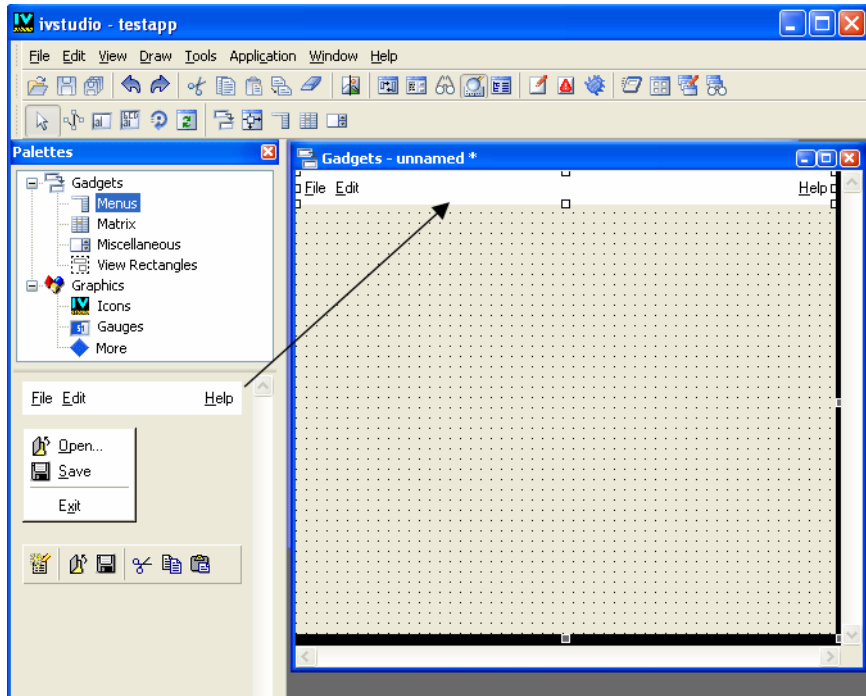
The Menu palette provides three types of menu gadgets that you can use in your panels. This section gives you information on how to create these objects.

- ◆ *Menu Bars*
- ◆ *Pop-up Menus*
- ◆ *Toolbars*

Menu Bars

To insert a menu bar (`IlvMenuBar`) in your panel:

1. In the top pane of the Palettes panel, click Menu.
- The Menu palette is displayed in the bottom pane of the Palettes panel.
2. Click the menu bar gadget and drag it to the active Gadgets buffer window.



The menu bar is automatically resized so it is as wide as the panel itself and default horizontal attachments are set to reflect panel changes (fixed, elastic, fixed).

To inspect this menu bar, double-click it or click the Inspect icon from the Main window toolbar. Its inspector looks like this:

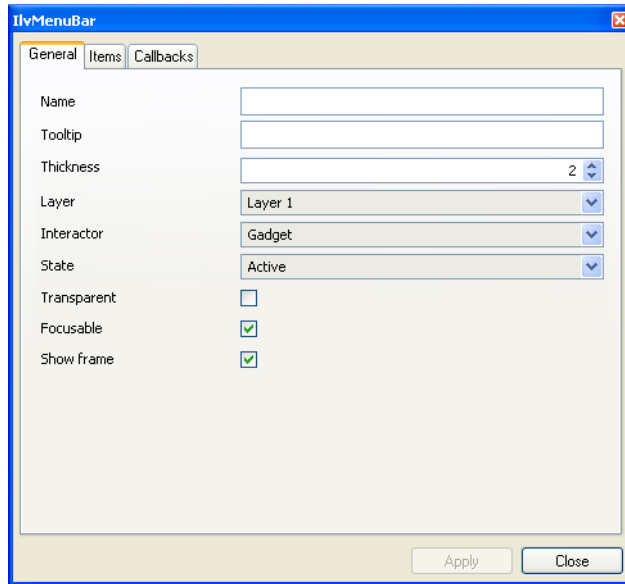


Figure 2.12 Menu Bar Inspector Panel (General page)

The options in the Items page let you insert, add, or remove items from the selected menu bar. You can also add a separator between a set of menu items or append a pop-up menu.

The left side of the page displays the structure of the menu bar as a tree. To apply changes to the whole menu bar or to any one of the items of which it is composed, select the appropriate item in the tree and make the changes you want in the right side of the page. Click Apply to validate the changes.

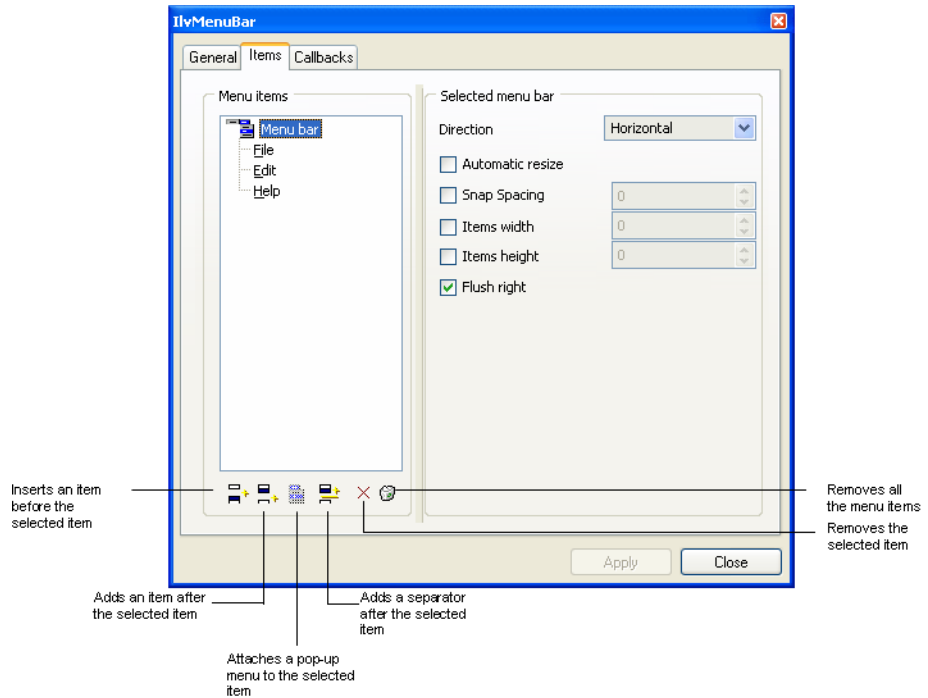


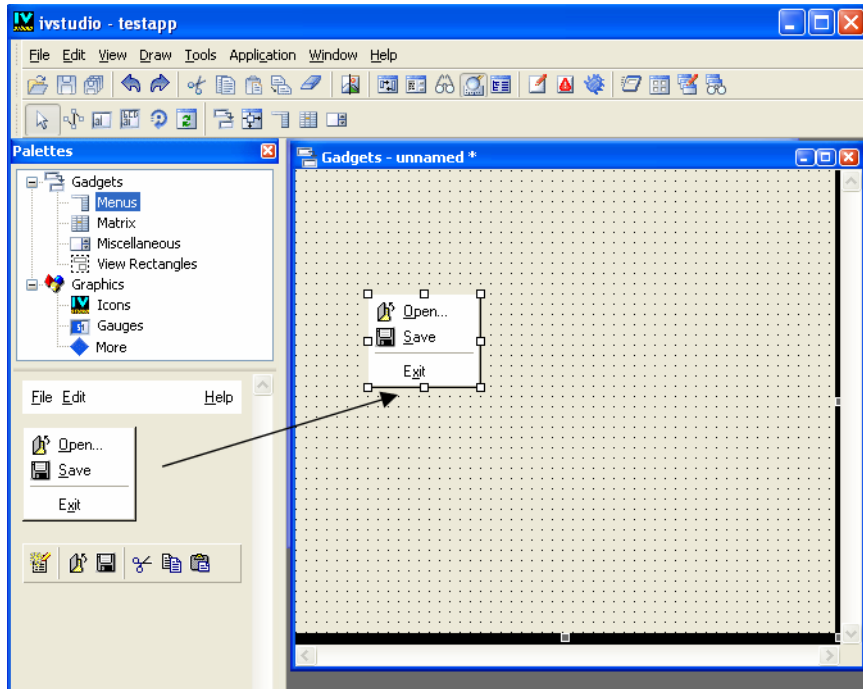
Figure 2.13 Menu Bar Inspector Panel (Items Page)

Pop-up Menu

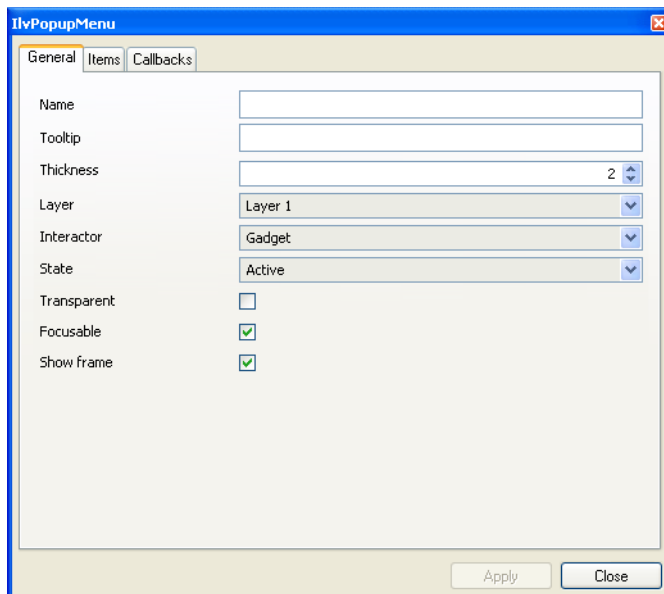
Before being attached to the menu bar, a pop-up menu must be inserted and edited in the workspace.

To insert a pop-up menu (`IlvPopupMenu`) in the workspace:

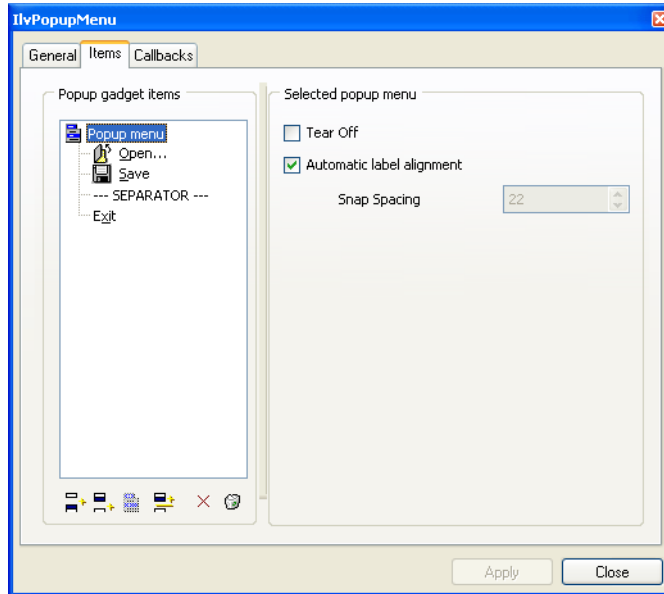
1. In the top pane of the Palettes panel, click Menus.
The Menus palette is displayed in the bottom pane of the Palettes panel.
2. Click the pop-up menu gadget and drag it to the Gadgets buffer window.



3. Double-click the pop-up menu to display its inspector panel.



4. To insert, add, or remove items from the pop-up menu in the buffer window, use the Items page of the PopupMenu inspector. You can also add a separator between a set of menu items.



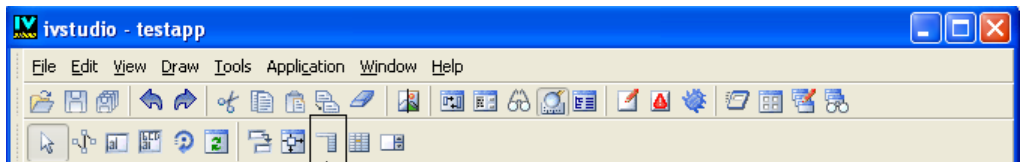
The left side of the page displays the structure of the pop-up menu as a tree. To apply changes to the whole pop-up menu or to each one of the items of which it is composed, select the appropriate item in the tree and make the changes you want in the right side of the page.

5. Click Apply to validate the changes.

Attaching Pop-up Menus to the Menu Bar

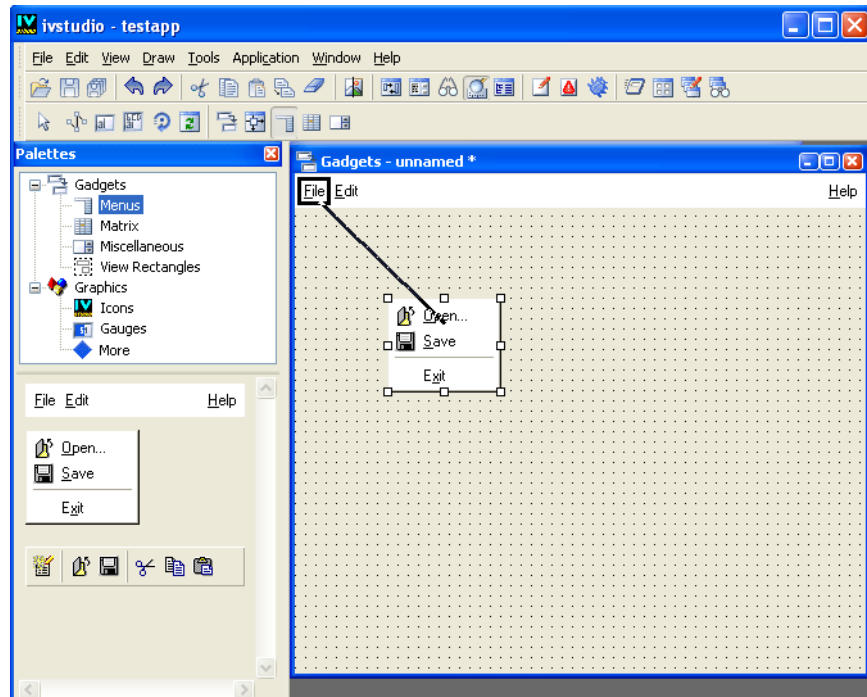
To attach pop-up menus to menu items:

1. Click the Menu mode icon in the Editing Modes toolbar:



Menu Mode Icon

- Click the pop-up menu and drag the mouse to the menu bar item to which you want to attach it.



As you drag the mouse, a black line appears linking the two items. The pop-up menu disappears when you release the mouse button.

To edit a pop-up menu that has been attached to a menu bar or another pop-up menu item:

- Go back to the Menu mode.
- Double-click the menu item to which the pop-up menu is attached.

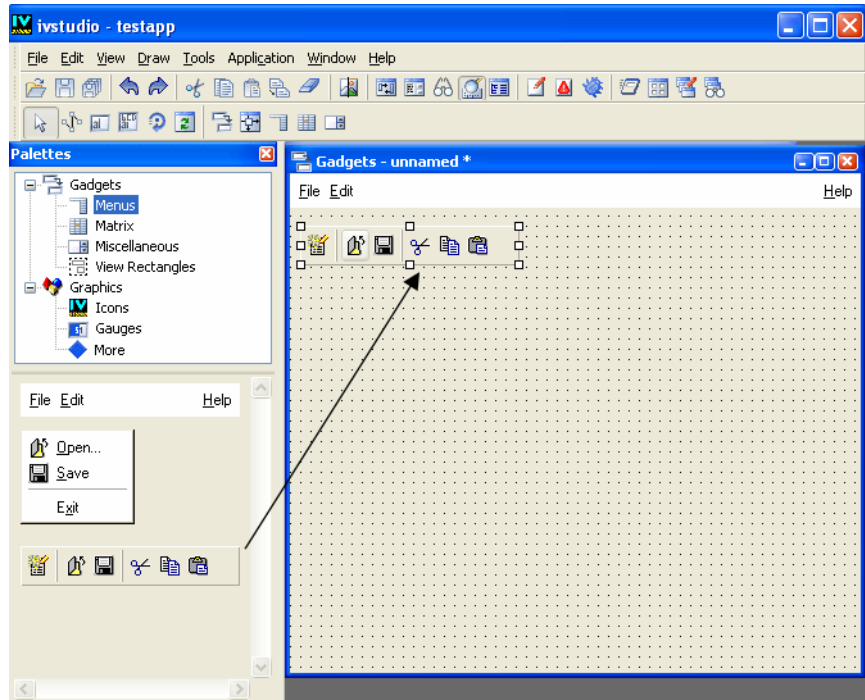
Its submenu tears off and can be selected and edited.

- To change a pop-up menu item, drag an object from the workspace and place it on the item to be changed. That object will then be removed from the workspace.
- To get a copy of the object used by a menu item, drag that item and place it outside the menu. You can then edit this copy and place it back in the menu item.

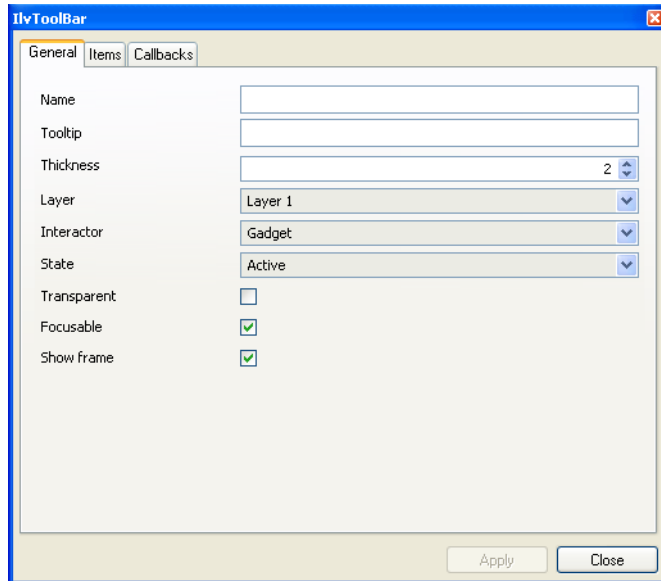
Toolbars

To insert a toolbar (`IlvToolBar`) in your panel:

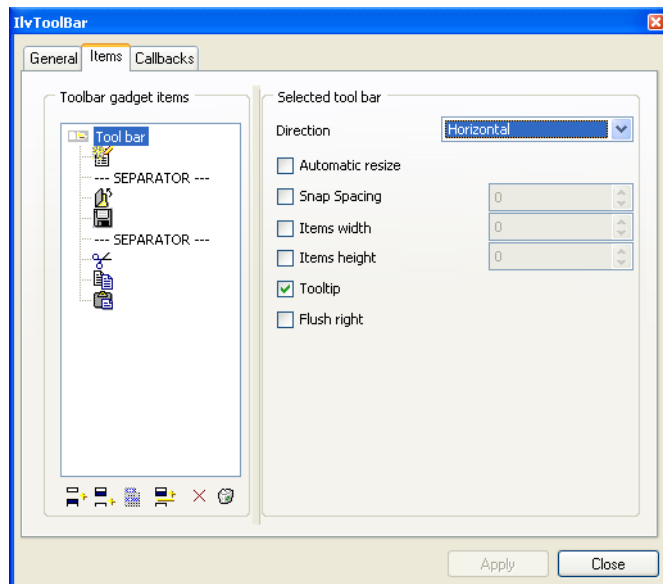
1. In the top pane of the Palettes panel, click **Menus**.
The **Menus** palette is displayed in the bottom pane of the Palettes panel.
2. Click the toolbar gadget and drag it to the **Gadgets** buffer window.



3. Double-click on the toolbar to display its inspector panel.



- To insert, add, or remove items from the selected toolbar, use the Items page of the Toolbar inspector. You can also add a separator between a set of toolbar items.



The left side of the page displays the structure of the toolbar as a tree. To apply changes to the whole toolbar or to any one of the items of which it is composed, select the appropriate item in the tree and make the required changes in the right side of the page.

Toolbars can be oriented horizontally or vertically. In addition, toolbar items can display tooltips and can be attached to pop-up menus.

Using Matrices

Use the Matrix mode to change items in your `IlvMatrix` or `IlvSheet` objects as well as in their respective inspector panels that let you edit their general properties.

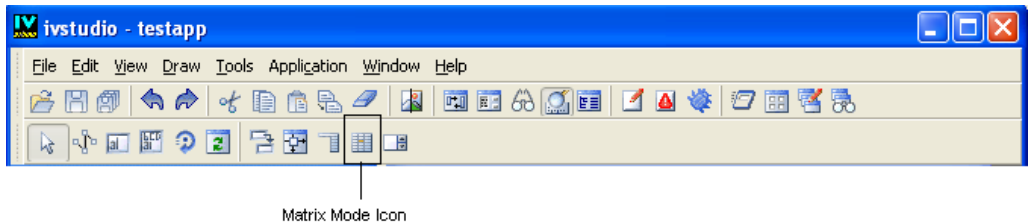
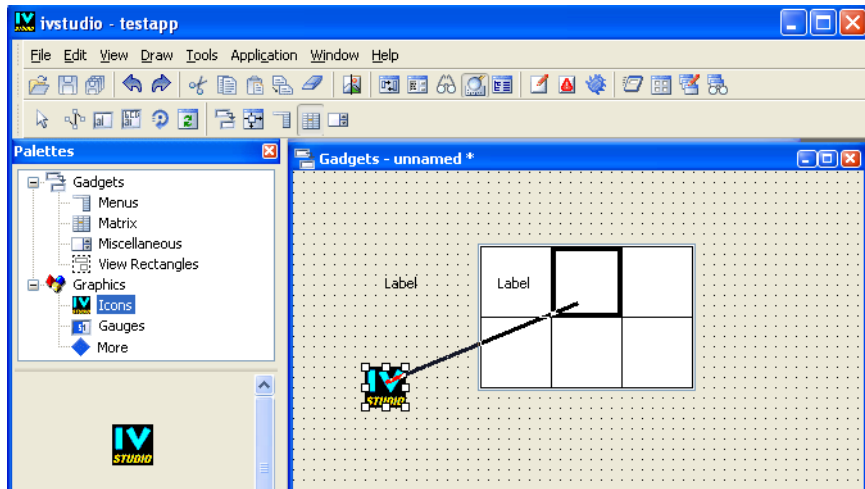


Figure 2.14 Matrix Mode Icon

Setting Up Matrix Items

You can set up a matrix item by dragging an object from the workspace and dropping it in the desired item: a copy of the dragged object is made and put in the matrix item; the source object remains available in the workspace.



If the dragged object is an `IlvLabel` object, the new matrix item becomes an `IlvLabelMatrixItem`. If the dragged object is an icon (of class `IlvIcon` or derived classes), the new item becomes an `IlvBitmapMatrixItem`. The matrix and item classes are documented in the IBM ILOG Views *Reference Manual*.

Extracting Matrix Items

You can extract the object from a matrix item by dragging that item and dropping it in the workspace. The extracted object can then be edited and put back where it was or copied to other items.

Inspecting Matrix Items

The Matrix mode provides you with an item inspector. To inspect a cell, double-click on it. The following inspector panel appears:

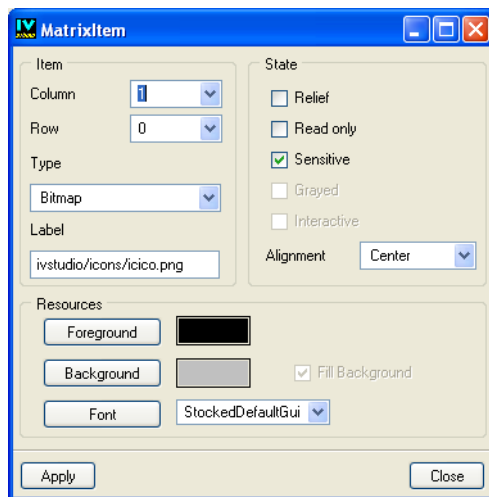


Figure 2.15 Matrix Item Inspector Panel

Item References

The Matrix Item inspector panel lets you inspect:

- ◆ one cell
- ◆ all the rows in a column
- ◆ all the columns in a row
- ◆ the whole matrix

The Column and Row fields display the coordinates of the inspected cell(s). If you want to inspect all the columns, enter “*” in the Column field. If you want to inspect all the rows, enter “*” in the Row field.

Item Type

The Type option menu lets you choose the matrix item class for the inspected cells:

Option	Matrix Item Class	Matrix Item Class with Resources
Empty	None. (Empty cells)	
Label	IlvLabelMatrixItem	IlvFilledLabelMatrixItem
Int	IlvIntMatrixItem	IlvFilledIntMatrixItem
Float	IlvFloatMatrixItem	IlvFilledFloatMatrixItem

Option	Matrix Item Class	Matrix Item Class with Resources
Double	IlvDoubleMatrixItem	IlvFilledDoubleMatrixItem
Bitmap	IlvBitmapMatrixItem	
Graphic	IlvGraphicMatrixItem	
Gadget	IlvGadgetMatrixItem	

The last column of the above table shows the matrix item classes that are used if you choose a foreground, a background or a font for your label or numeric items. For more information, see the sections in the *IBM ILOG Views Reference Manual* corresponding to the classes in the above table.

Item Flags

Use the Sensitive, Read only, Relief and Interactive toggle buttons to set the corresponding flags for your items:

Toggle button	See class	get function	set function
Sensitive	IlvMatrix	isItemSensitive	setItemSensitive
Read only	IlvMatrix	isItemReadOnly	setItemReadOnly
Grayed	IlvMatrix	isItemGrayed	setItemGrayed
Relief	IlvMatrix	isItemRelief	setItemRelief
Interactive	IlvGadgetMatrixItem	isInteractive	setInteractive

Item Resources

Use the Foreground, Background, and Font fields to set the colors and fonts for the selected items.

Validating

Click Apply to validate the characteristics you edit in the Matrix Item inspector panel.

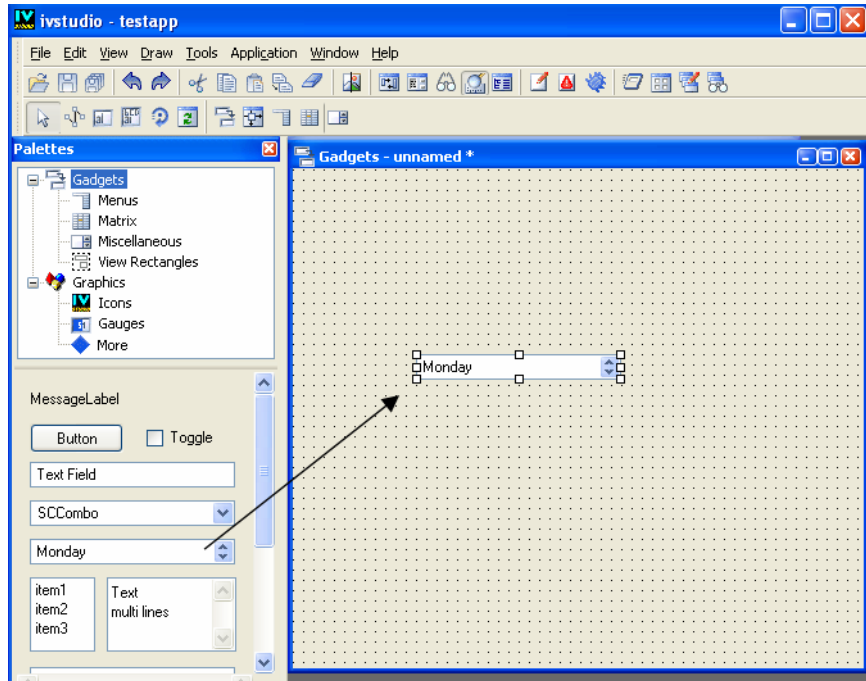
Editing Spin Boxes

To include a spin box in your panel, you will need to insert the spin box and then specify the type of item you want to appear in the spin box.

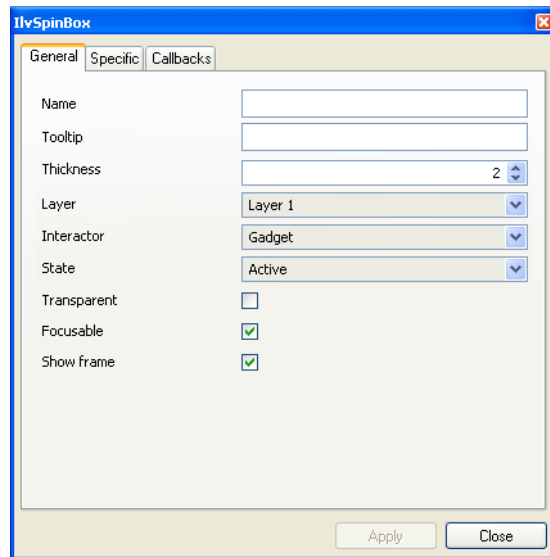
Inserting a Spin Box

To insert a spin box (IlvSpinBox) in your panel:

1. In the top pane of the Palettes panel, click Gadgets.
The Gadgets palette is displayed in the bottom pane of the Palettes panel.
2. Click the spin box gadget and drag it to the Gadgets buffer window.



3. Double-click the spin box gadget to display its inspector.
4. Use the Spin Box inspector to edit the items that appear in the spin box.




The Specific page of the inspector allows you to add fields to the spin box, specify the values that appear in the fields and specify how the spin arrows appear in the spin box.

Setting the Type of Spin Box Item

The default item in the spin box gadget is an `IlvTextField` object. By using the Spin Box editing mode, you can specify the type of gadget object that appears as a spin box item. For example, you may want to have an `IlvNumberField` as the spin box item rather than an `IlvTextField`.

To set an `IlvNumberField` as the item in a spin box, do the following:

1. Drag a spin box gadget from the Palettes panel to the Gadgets buffer window.
2. Double-click the spin box gadget to display the Spin Box inspector.
3. Click the Specific tab to display the Specific page.
4. Select the `IlvTextField` item in the Fields box.
5. Click the Remove icon  below the Fields box.

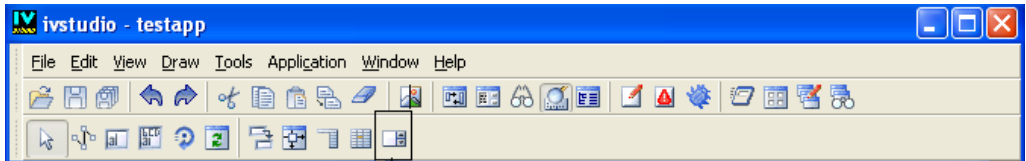
The item is removed along with its settings defined in the other fields of the inspector.

6. In the upper pane of the Palettes panel, click Miscellaneous.

The Miscellaneous palette is displayed in the bottom pane of the Palettes panel.

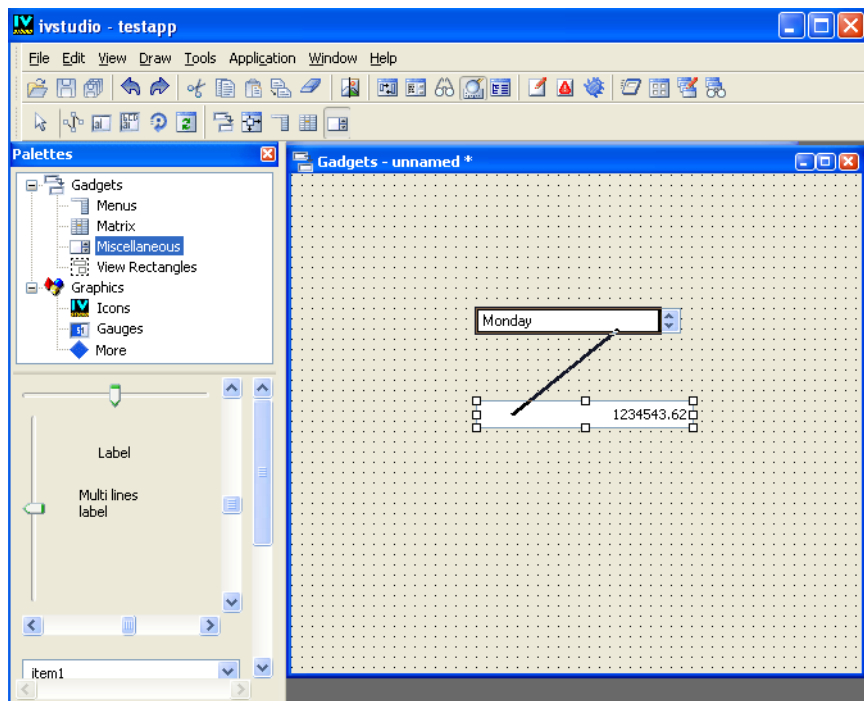
7. Click the `IlvNumberField` gadget and drag it to the Gadgets buffer window.

8. Click the Spin Box icon in the Editing Modes toolbar.

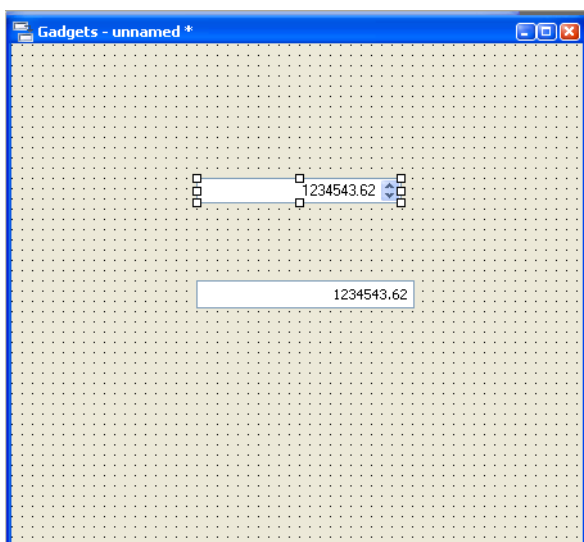


Spin Box Icon

9. Drag a line from the number field gadget to the spin box gadget.



10. The spin box now contains the number field and you can specify the settings for the item in the spin box inspector.



Editing Applications

This chapter describes how to work with application buffers. Application buffers contain panel instances that are derived from panel classes. Panel classes and panel instances are created and handled using a special palette called the Panel Class palette.

You will find information on the following topics:

- ◆ *The Application Buffer*
- ◆ *Application Description File*
- ◆ *Other Generated Files*
- ◆ *The Application Inspector*
- ◆ *Editing an Application*

The Application Buffer

In IBM® ILOG® Views Studio, you edit an application via the Application buffer window. When you launch IBM ILOG Views Studio, a default application called “testapp” is created. To activate the Application buffer window, choose <Application> from the Window menu or click in the Application buffer window (by default, an empty Application buffer window is displayed at start-up). Only one application can be edited at a time.

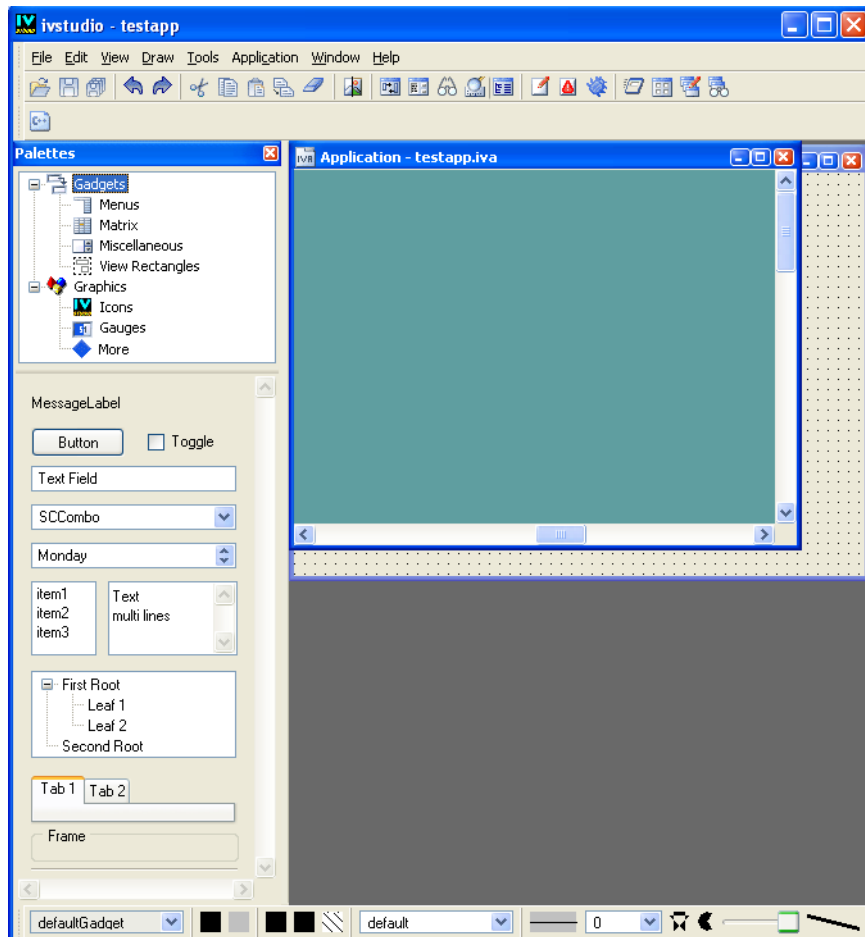


Figure 3.1 The Application Buffer Window in the Main Window

When the Application buffer window is activated, you will notice the following:

- ◆ The title bar of the Main window changes to reflect the application file name, followed by the word <Application>.
- ◆ The type of the current buffer, Application, is displayed at the bottom right of the Main window.
- ◆ The Editing Modes toolbar that appears at the top of the Main window contains a single icon, corresponding to the Generate command.
- ◆ The generic inspector disappears.

When you edit an application, you will use the Application buffer window along with the Panel Class palette. To activate the Application buffer window and the Panel Class palette, click the Edit Application icon in the toolbar at the top of the Main window.

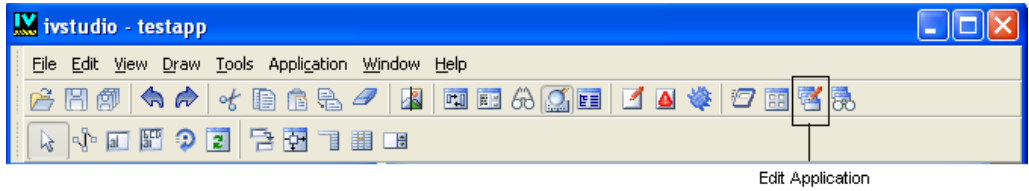


Figure 3.2 The Edit Application Icon

When you edit an Application buffer, the Main window should look something like this:

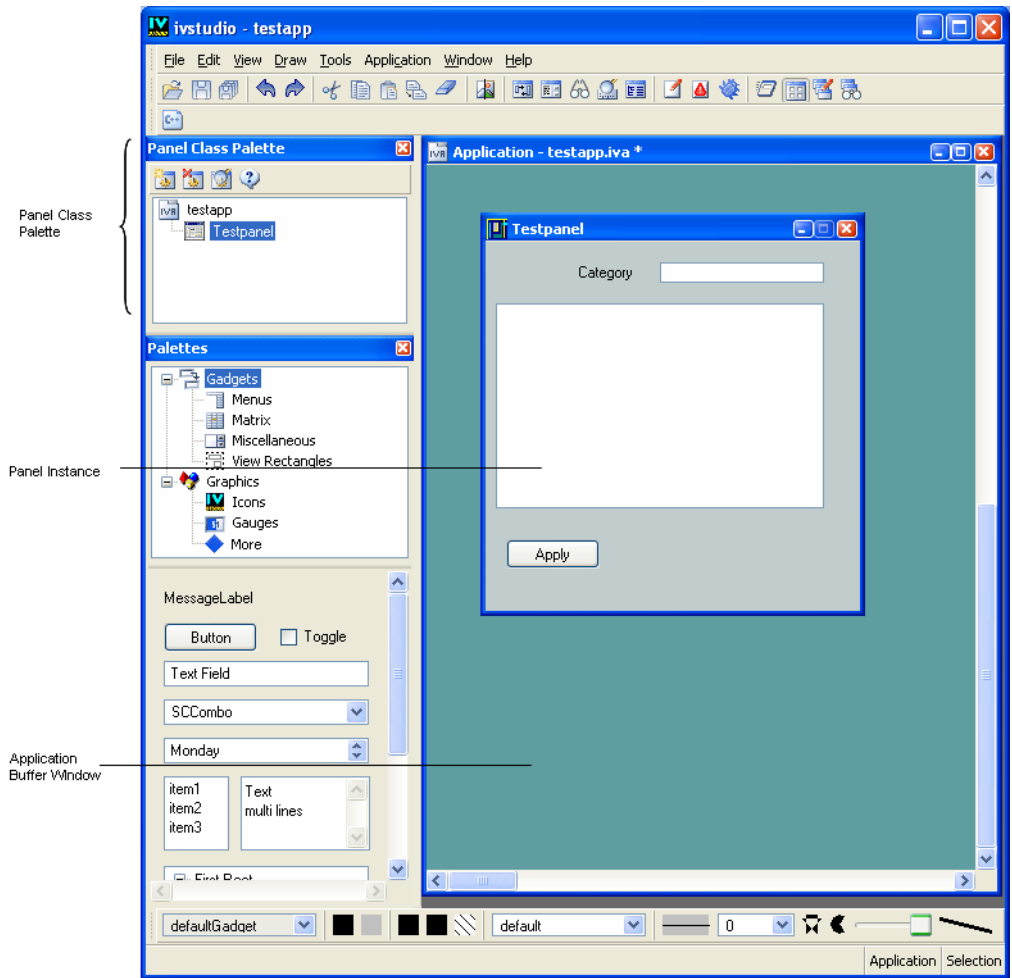


Figure 3.3 *Editing an Application Buffer*

The Application buffer window contains any panel instances that have been added to it.

The Panel Class palette lets you create new panel classes, as well as remove and inspect them. The icons in the palette show the panel classes that have been created for the current application.

Application Description File

The properties of the application, and also panel classes and panel instances that make up the application, are saved in a data file that typically has an `.iva` extension. Although the file format is easy to understand, it is better to use IBM® ILOG® Views Studio to edit the application buffer.

Only one application can be edited at a time. If you are editing an application and want to open a new one, save the current application and then create a new application or load a previously saved application.

You can use the following commands in the File menu to work with application description files.

New > Application


By default, IBM ILOG Views Studio reates an empty application when it is launched (“testapp”). Choose <Application> from the Window menu to start editing “testapp”, or click on its window to activate it.

If you are already editing an application, and want to create a new one, save your current application and choose New from the File menu and then Application in the submenu that appears.


Save As...

Before saving an application, make sure that the current buffer is the Application buffer (using the Window menu, if necessary). To save a new application for the first time, choose Save As... from the File menu. This command opens the File Chooser that lets you save the application description file in a directory. Application files are saved with the `.iva` file extension.

Save

To save an application, choose Save from the File menu or click the Save icon  from the toolbar. This command saves the application description in a data file. See the Save As... command above for saving a new application file.

Open...

To load an application previously saved by IBM ILOG Views Studio, choose Open... from the File menu, or click the Open icon  from the toolbar. This command opens a File Chooser that lets you choose an applicaon file. To filter the list of files that are displayed in the File Chooser, select Application files in the File type option menu.

This command discards the current application; so, if necessary, save the current application first.


Other Generated Files

In addition to the data files, IBM® ILOG® Views Studio generates the following for each application:

- ◆ A header file and a source file for the generated C++ application class,
- ◆ A header file and a source file for the panel class corresponding to each buffer,
- ◆ A simple `make` file.

The location of these files can be individually specified and, in each C++ generated file, you can insert your own code using special inspector panels.

The Application Inspector

The application properties can be displayed and edited using the Application inspector. To display the inspector of the current application, choose Application Inspector from the Application menu in the Main window or click the Application Inspector icon  in the Main window toolbar.

The Application inspector is opened for the current application. This may be the default application if you have not already opened an application.

The Application inspector has five notebook pages: General, Options, Header, Source, and Script and four buttons: Apply, Reset, Close, and Help.

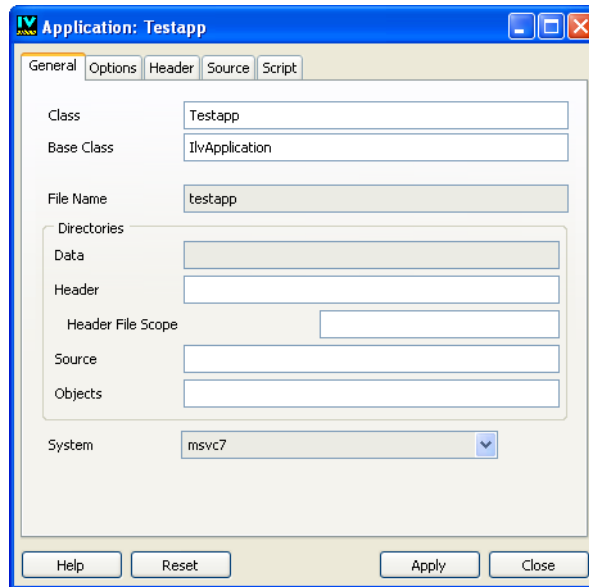


Figure 3.4 General Page of the Application Inspector

The General Page

The General page of the Application inspector contains the following fields:

Class The class name of the generated application can be specified in the Class field. By default it is `Testapp`. The name that you specify in this field must be a valid C++ class name.

Base Class The base class of the application can be specified in the Base Class field. By default it is `IlvApplication`. The name that you specify in this field must be a valid C++ class name.

Instead of deriving the generated class from `IlvApplication`, you can specify your own base class. In this case, the given class must be derived from `IlvApplication` and must include compatible constructors. Of course, the declaration of your base class must be known by the compiler when compiling the generated files; that is, it must be inserted or included in the generated file. See *The Header and Source Pages* on page 93.

File Name Shows the name of the `.iva` file containing the application.

Directories

Data The Data field displays the directory where the application data file is saved. This field cannot be edited. To change the location of the application data file, activate the

Application buffer window, choose Save As... from the File menu of the Main window, and save it in the directory of your choice.

Header Use the Header field to specify where the application header file should be generated. By default, the header file is generated in the directory where the application data file is saved. This directory is relative to the application data file directory.

Header File Scope The directory where header files are generated is obtained by appending a header file scope to the specified header directory. The option Header File Scope is used to specify a subdirectory that is generated in the `#include` statements.

Assuming that the application file is in the directory `/myappdir`, the Header directory is `include`, and the Header File Scope is `myapp`, header files are generated in the directory `/myappdir/include/myapp`. The generated `#include` statements corresponding to the application header files are the following:

```
#include <myapp/file1.h>
#include <myapp/file2.h>
```

instead of

```
#include <file1.h>
#include <file2.h>
```

Source Use the Source field to specify where the application source file should be generated. By default, the source file is generated in the directory where the application data file is saved. This directory is relative to the application data file directory.

Objects Use the Objects field to specify the location where the application `make` file is generated. By default, the `make` file is generated in the directory where the application data file is saved. This directory is relative to the application data file directory.

System Use this menu to specify the name of the platform for which you want to generate the `make` file. The default platform is the one on which IBM ILOG Views Studio is running.

Motif This toggle button is only visible if the platform you choose in the System option menu is an X11 platform. If this toggle button is selected, the generated `make` file chooses the Motif version of the IBM ILOG Views libraries and links the `libXt` and `libXm` libraries to your application.

The Options Page

The Options page of the Application inspector is illustrated below:

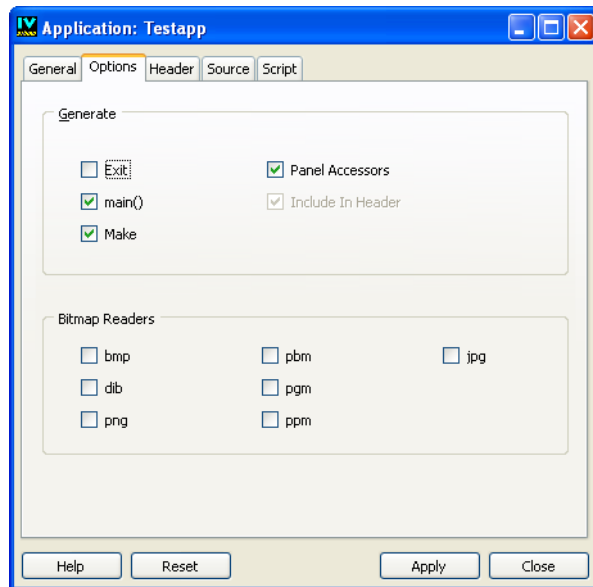


Figure 3.5 Options Page of the Application Inspector

The Options page contains the following fields:

Generate

Exit An Exit button, in a separate panel, can be activated when you run the generated application. This provides you with a simple way to quit the application. Select this toggle button if you want to set the Exit button.

main() Select this toggle button if you want IBM ILOG Views Studio to generate a simple main function in the application source file. If the generated application class is `Testapp`, and the application file base name is `myappli`, the main function looks like this:

```
main(int argc, char* argv[])
{
    Testapp* appli = new Testapp("myappli", 0, argc, argv);
    if (!appli->getDisplay())
        return -1;
    appli->run();
    return 0;
}
```

Make Select this toggle button if you want a simple make file to be generated.

Panel Accessors A panel accessor is a member function of the generated application that lets you access a particular panel of your application. If you check the Panel Accessors toggle button, IBM ILOG Views Studio generates a member function for each panel instance. The member function has the following signature:

```
MyPanelClass* getMyPanelInstance() const;
```

where `MyPanelClass` is the type of panel instance named `MyPanelInstance`. The names of your panels must be valid C++ names.

Include in Header In the generated code of an application, the header files generated for the panel classes of the application must be included. The necessary `#include` statements can be generated in the application header file or in the application source file. If you want to generate the panel accessors (the Panel Accessors toggle button is checked), the headers of the panel classes need to be included in the application header file. In this case, the Include In Header toggle button is unavailable, since you have no choice. Otherwise, the `#include` statements can be generated in the application source file instead of the application header file. To minimize the compilation dependencies of your whole application, do not check this toggle button.

Bitmap Readers

The bottom part of the Options page contains information on bitmap readers. The toggle buttons in this part of the inspector let you explicitly register predefined bitmap readers in the generated code.

The Header and Source Pages

The Header and Source pages can be used to add code to the header and source files.

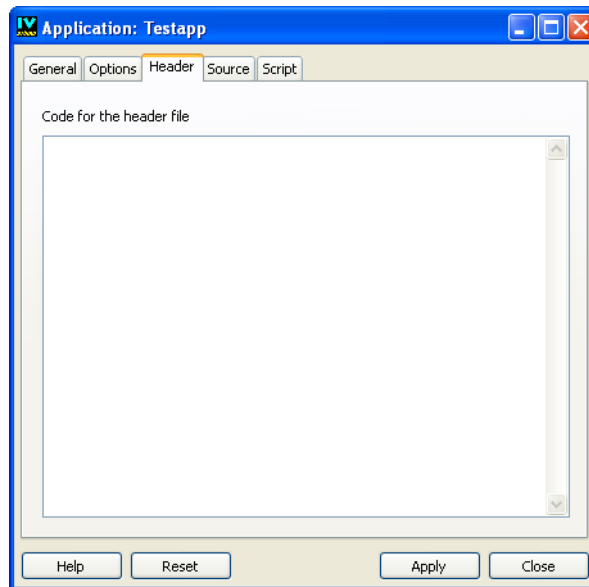


Figure 3.6 Header Page in the Application Inspector

Code for the header file The text you enter in this page is inserted as typed in the Application header buffer, after the generated `#include` statements and before the declaration of the generated class. If you want to subclass the generated class from a class other than `IlvApplication`, you have to insert the `#include` statement to include the file declaring your base class. Of course, instead of inserting code, you can use this feature to comment your application.

Code for the source file The text you enter in this panel is inserted as typed in the application source file just before defining the generated member functions. You can use this text to comment the generated file or to insert any C++ code.

The Script Page

The Script page can be used to specify the use of IBM ILOG Script.

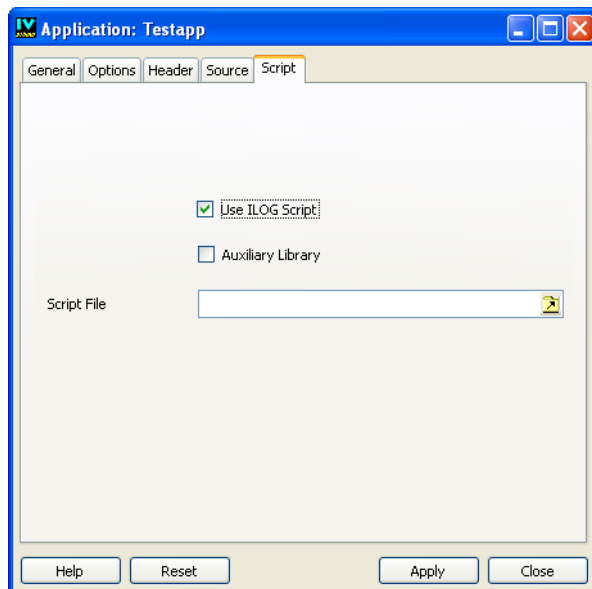


Figure 3.7 Script Page in the Application Inspector

The Script page contains the following fields:

Use IBM ILOG Script Select this toggle button if you want to use IBM ILOG Script.

Auxiliary Library Select this toggle button if you want to use the auxiliary library of IBM ILOG Script for IBM ILOG Views in your application through the scripting language. This library lets you use additional features, such as the dialog boxes. For more information, see the chapter “IBM ILOG Script *Programming*” in the IBM ILOG Views *Foundation User's Manual*.

Script File Enter the name of a file containing script code or click the button next to the text field to display a File Chooser to select a file.

The Application Inspector Buttons

These buttons appear at the bottom of the Application inspector.

Apply Applies the changes made to the application properties.

Reset Resets the application properties to their initial values.

Close Closes the Application inspector.

Help Displays online help about the fields in the Application inspector.

Editing an Application

In IBM® ILOG® Views Studio, you edit an application using the Panel Class palette. Panel classes can be added to the Panel Class palette and then dragged to the Application buffer window to create panel instances. A panel instance appears as it will look in the final application. The dimensions and the position of the panel can be directly edited within the Application buffer window.

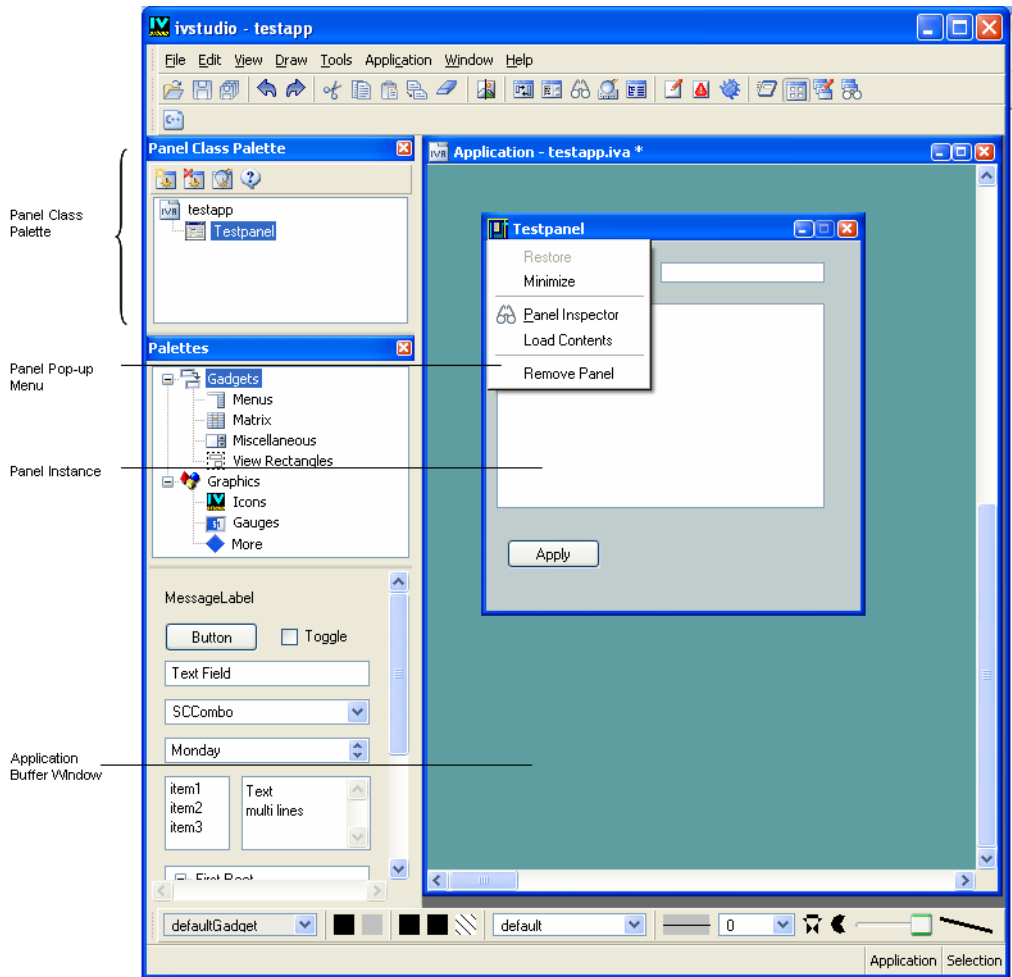




Figure 3.8 Panel Class Palette

The following sections explain how to create panel classes and add instances of these panel classes to your application.

The Panel Class Palette

The Panel Class palette is used to create, inspect, or remove panel classes. This palette can be accessed, whatever the current buffer, by clicking the Panel Class Palette icon  in the Main window toolbar or by selecting Panel Class Palette from the Code menu. It can also be opened together with the Application buffer by choosing the Edit Application icon  from the toolbar of the Main window.

The Panel Class palette consists of a toolbar containing the commands that can be used to manipulate panel classes, and a panel class buffer that shows icons representing the existing panel classes.

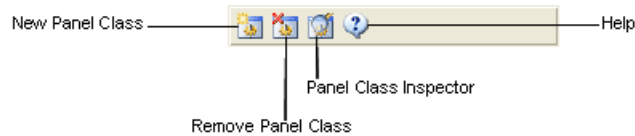


Figure 3.9 *Toolbar of the Panel Class Palette*

You can do the following using the commands in the Panel Class palette toolbar:

- ◆ **New Panel Class** Creates a new panel class from the current buffer. The current buffer must have already been saved.
- ◆ **Remove Panel Class** Removes the selected panel class from the palette.
- ◆ **Panel Class Inspector** Opens the inspector of the selected panel class.
- ◆ **Help** Lets you access Online Help on the Panel Class palette.

When you double-click a panel class icon in the Panel Class palette, the file containing the panel description is opened and set as the current buffer.

When you double-click the background of the Panel Class palette (without clicking a panel class icon), the Application buffer is set as the current buffer.

Panel Classes

Panel classes can be created using the Panel Class palette. These classes can then be used to add panel instances to the Application buffer.


For each Gadgets buffer that is part of the edited application, IBM ILOG Views Studio generates a C++ class derived from `IlvGadgetContainer`. The generated class does the following:

- ◆ Reads the data used to create the panel objects.
- ◆ Generates callbacks as methods.
- ◆ Generates accessors for named objects.

Adding a Panel Class

To add a new panel class to the application:

1. Make sure that the required panel buffer is open, and is the current buffer.

2. Click the Panel Class Palette icon  in the Main window toolbar to open the Panel Class palette.

3. Click the New Panel Class icon  in the Panel Class palette toolbar.

The new panel class is added to the palette.


Removing a Panel Class

To remove a panel class from the application:

1. In the Panel Class palette, select the panel class you want to remove.

2. Click the Remove Panel Class icon  in the Panel Class palette toolbar.

The Panel Class Inspector

To inspect a Panel class, click the Panel Class Inspector icon  in the Panel Class palette toolbar.

The Panel Class inspector appears:

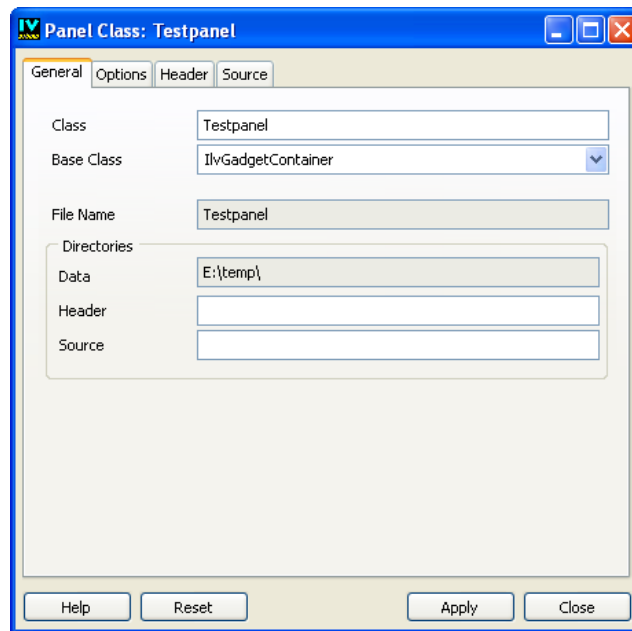


Figure 3.10 Panel Class Inspector

There are four notebook pages in the Panel Class inspector, each containing a set of properties of the inspected panel class.

The General Page

Class Use this field to name the C++ panel class. The class name must be a valid C++ class name. By default, IBM ILOG Views Studio names this class by capitalizing the first letter of the corresponding buffer name.

Base Class Use this field to specify the base class for the class generated. By default for gadget buffers, IBM ILOG Views Studio derives the generated class from `IlvGadgetContainer`.

Instead of deriving the generated panel class from `IlvGadgetContainer`, you can specify your own base class. In this case, the given class must be derived from `IlvGadgetContainer` and include compatible constructors. Of course, the compiler must know the declaration of your base class when compiling the generated files, so it must be inserted or included in the generated file. See *The Header and Source Pages* on page 101.

File Name This field shows the name of the file that contains the selected panel class. It cannot be edited.

Data This field displays the directory where the panel data file (.ilv file) is saved.

Header Use this field to specify the directory where the panel class header file is generated. If this field is empty, the header file is generated in the same directory as the application header file.

Source Use this field to indicate the directory where the panel class source file is generated. If this field is empty, the source file is generated in the same directory as the application source file.

The Options Page

The Options page of the Panel Class inspector is illustrated below.

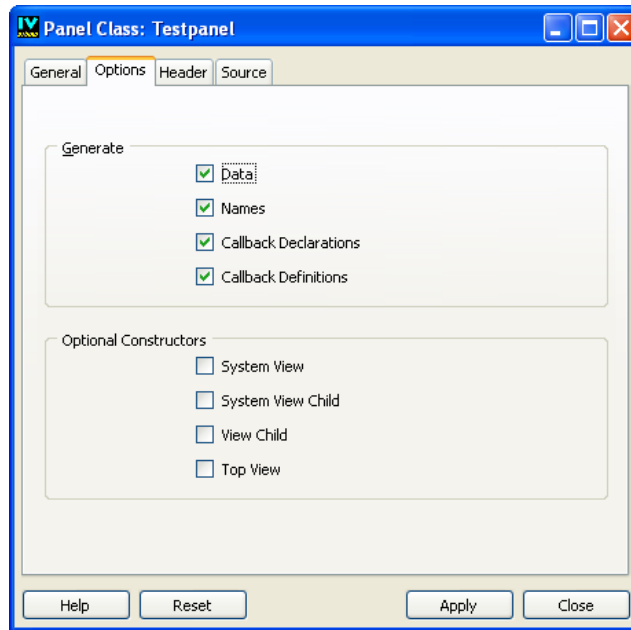


Figure 3.11 Options Page of the Panel Class Inspector

The Generate section contains the following:

Data Select this toggle button to have IBM ILOG Views Studio generate the data string in the C++ panel class code so that its constructor does not need to read the data file at runtime. The code generated with data is used only on the UNIX® platforms. On Windows® systems, the string data generated is not used.

Names Select this toggle button to have IBM ILOG Views Studio generate member functions that return the named objects in your panel. For example, if you have a text field named `MyTextField` in your panel, the following member function is generated:

```
IlvTextField* getMyTextField() const
{ return (IlvTextField*)getObject("MyTextField"); }
```

The generated member functions are always named following this rule.

Callback Declarations IBM ILOG Views Studio provides you with a simple way to deal with callbacks. When the Callback Declarations toggle button is selected, it generates an `IlvGraphicCallback` function and declares a default virtual member function. The generated `IlvGraphicCallback` invokes the corresponding virtual member function, which has the same name as the callback you specified in IBM ILOG Views Studio. Therefore, the names you use for callbacks must be valid C++ function names.

The Callback Declarations toggle button must be selected for the Callback Definitions toggle button to have any effect.

Callback Definitions The default definition code (the body function) for these callback virtual member functions can be generated, letting you test your application before defining the real callbacks. When you select the Callbacks Definitions toggle button, you redefine your own versions of the callbacks in your derived classes.

If you do not want these function definitions to be generated, do not select the Callback Definitions toggle button. This is useful if you do not want to derive a class from the generated class. In this case, you can write your own definition of these member functions in a separate file that will not be erased by future code generations.

The callback registering task is generated in the C++ code so that you only have to define the callback methods.

The Optional Constructors section has two toggle buttons that let you generate your panel class to use within the native system views.

System View Select this toggle button if you want to create a panel by using an existing system view.

System View Child Select this toggle button if you want to create your panel as a child window of an existing system view.

The Header and Source Pages

To insert your own code in the generated panel class header or source files, click the Header or Source page of the Panel Class inspector.

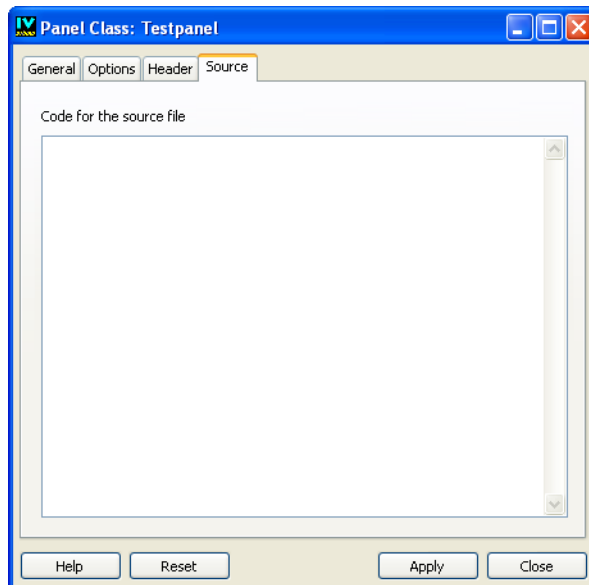


Figure 3.12 Source Page in the Panel Class Inspector

Code for the header file The text you enter in this field is inserted as typed in the panel class header file, after the generated `#include` statements and before the declaration of the generated class. If you want to derive the generated class from a class other than `IlvGadgetContainer`, you have to insert the `#include` statement to include the file declaring your base class. Of course, instead of inserting code, you can use this feature to comment your panel class.

Code for the source file The text you enter in this field is inserted as typed in the panel class source file just before defining the generated member functions. You can use this text to comment the generated file or to insert any C++ code.


Panel Instances

Once the panel classes have been defined in the Panel Class palette, you can create and inspect the instances of these classes.

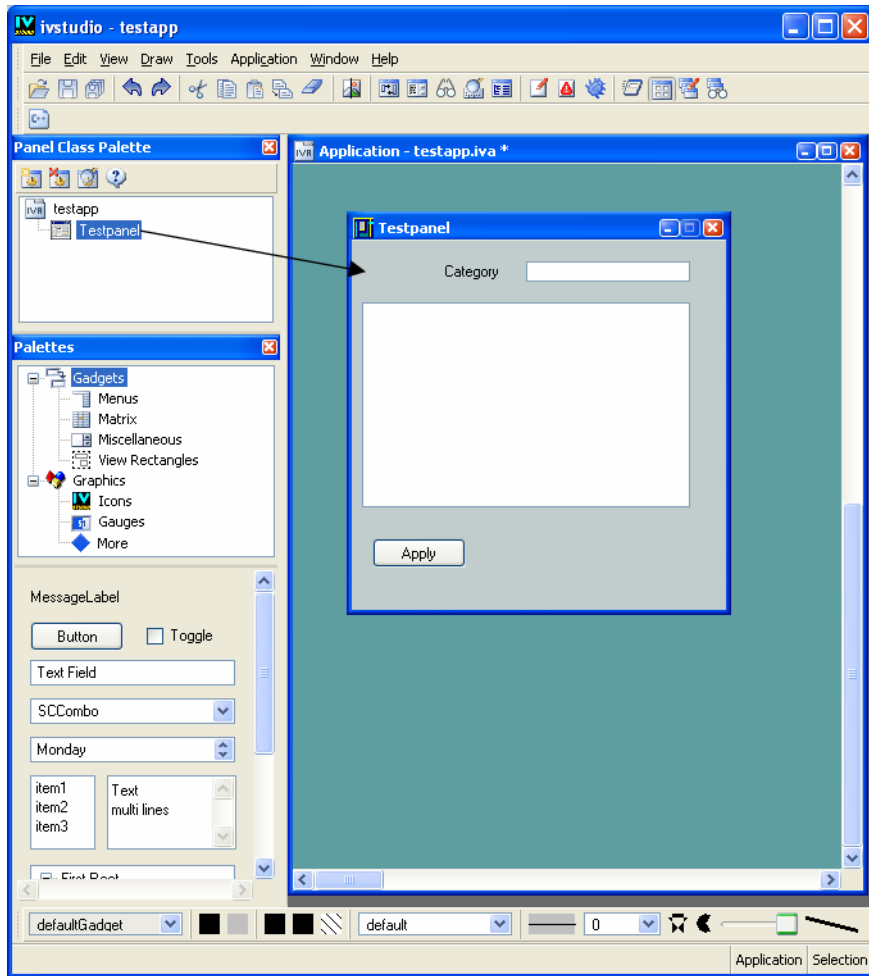
Adding a Panel to an Application

To add a panel to your application:

1. Make sure that the Application buffer window is the current window, and that the Panel Class palette is displayed.

To display the Panel Class palette, click the Panel Class Palette icon  in the Main window toolbar.

2. In the Panel Class palette, select the panel class icon and drag it directly into the Application buffer window.



An instance of the panel is created in the Application buffer window. This instance is represented as a window. The panel class name is the default name of your new panel.

Managing Panel Instances in the Application Buffer

Once panel instances have been added to your Application buffer window, you can manage them in the same way that you manage windows in a windowing environment. Each panel instance window has a pop-up menu that appears when you click on the top-left corner of the window. The menu has standard window options, such as Restore and Minimize.

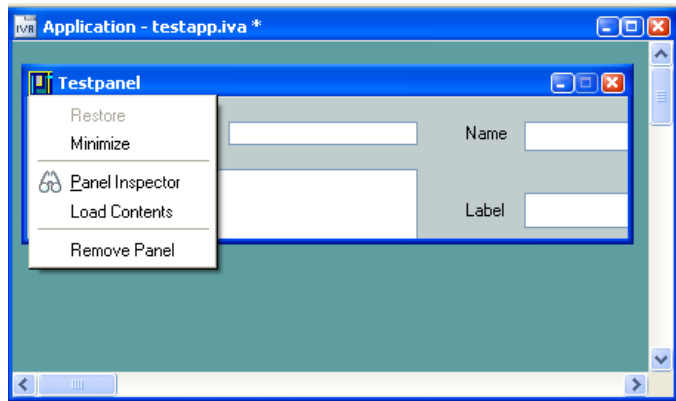


Figure 3.13 Panel Pop-up Menu

The Restore menu item enables you to restore a minimized panel instance to a window with its original size.

The Minimize menu item reduces the panel instance to its title bar.

A panel instance can be inspected by choosing Panel Inspector from this menu.

The Load Contents menu item lets you explicitly load the contents of your panel instance. This is useful when you use the `noPanelContents` option. (See the description of the `noPanelContents` option in the section “Configuration Options for the Gadgets Extension” in Chapter 5, *Customizing the Gadgets Extension of IBM ILOG Views Studio*.)

The Remove Panel option removes the panel instance from the Application buffer window.

Inspecting a Panel Instance

To inspect a panel instance:

1. Click the box in the top-left corner of the panel instance.

A pop-up menu appears.

2. From that menu, choose Panel Inspector.

You can also double-click the title bar of the panel instance window.

The Panel Instance inspector appears:

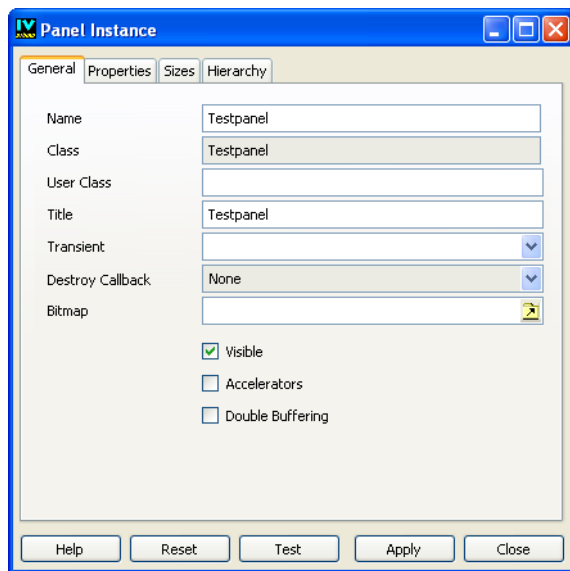


Figure 3.14 The General Page of the Panel Instance Inspector

The four notebook pages of the Panel Instance inspector allow you to edit the properties of your panel instances.

The General Page

Name Use this field to name your panel. This name must be a valid C++ name if you want IBM ILOG Views Studio to generate the panel accessors for your application.

Class This text field displays the name of the generated class of your panel. You cannot edit this field.

User Class If you use a class that is derived from the generated class displayed in the Class field, type its name in this field. In this case, the file declaring such a class must be included in the generated application class file (see *The Header and Source Pages* on page 93) and its definition module must be linked to the final application.

Title Use this field to set the title of your panel.

Transient Use this option menu to set up a relationship between two panels. By selecting an existing panel in this field, you are specifying that the current panel will always be displayed in front of the panel selected in the Transient field.

Destroy Callback This is the callback invoked when the panel is closed by the window manager. Use the option menu to choose a default destroy callback for your panel.

Bitmap Enables you to specify a bitmap as the panel background.

Visible If you do not want the panel to be displayed when launching the application, do not select the Visible toggle button. By default, the panel is visible.

Accelerators If checked, the panel instance is created with the default container's accelerators. This means that the panel constructor is called with `useacc` parameter set to `IlvTrue`.

Double Buffering If checked, the inspected panel uses the double-buffering mechanism. This generates the following code:

```
cont ->setDoubleBuffering(IlvTrue)
```

The Properties Page

The toggle buttons in the Properties page let you specify the window frame properties. In the generated code, the selected options are combined to set the `properties` parameter in the call to the panel constructor. Each option corresponds to a predefined property.

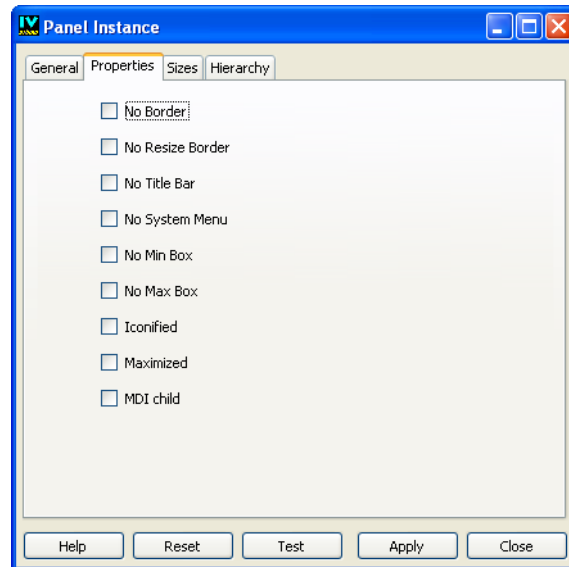


Figure 3.15 The Properties Page of the Panel Instance Inspector

The following table shows the predefined properties that are linked to the toggle buttons in the Properties page of the Panel Instance inspector.

Toggle button	Predefined Property
No Border	<code>IlvBorder</code>
No Resize Border	<code>IlvNoResizeBorder</code>

Toggle button	Predefined Property
No Title Bar	IlvNoTitleBar
No System Menu	IlvNoSysMenu
No Min Box	IlvNoMinBox
No Max Bars	IlvNoMaxBox
Iconified	IlvIconified
Maximized	IlvMaximized
MDI child	IlvMDIChild

The Sizes Page

The Sizes page of the Panel Instance inspector has three sections: Bounding Box, Minimum Size, and Maximum Size.

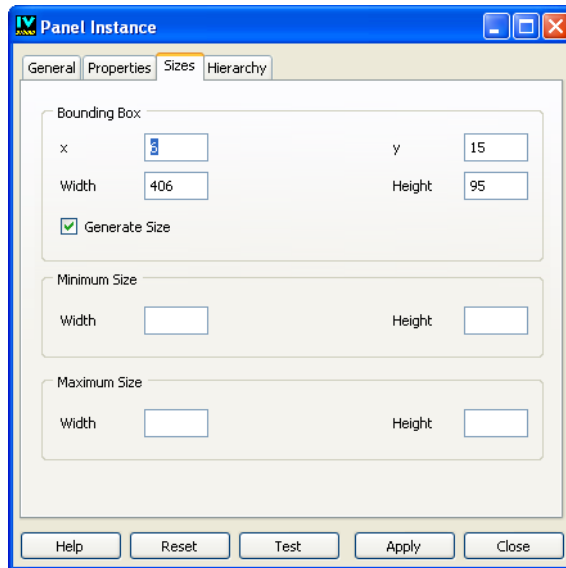


Figure 3.16 *The Sizes Page of the Panel Instance Inspector*

x, y, Width, Height Use these fields to specify the initial position of the panel. The panel size defaults to the size of the buffer. If you want to assign a new size to your panel, turn on the toggle button **Generate Size** and enter desired values in the **Width** and **Height** fields.

Maximum and Minimum Size Use these fields to specify the Maximum or Minimum field sizes for the panel by entering the desired values in the **Width** and **Height** fields

Panel Instance Buttons

Apply Click to validate your panel options.

Reset Click to reset the Panel Instance inspector to the last validated values.

Test Click to test your panel. Unlike the global test for the application, only the inspected panel is shown by this action. The options entered but not yet validated with the Apply button are used to create the test panel. Even if the panel is configured to be not visible, it can be tested.




Close Click to close the Panel Instance inspector.


Help Click to access Online Help.

Editing Subpanels

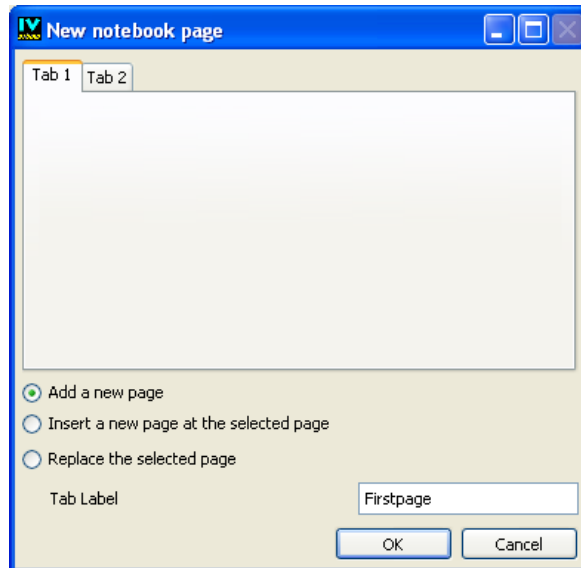
You can use the Application buffer window to create panel instances that are contained in an `IlvContainerRectangle` object or in an `IlvNotebook` object. In other words, container rectangles and notebooks can hold subpanels.

To make a panel instance a subpanel of a container rectangle or of a notebook, proceed as explained below:

1. Click View Rectangles in the top pane of the Palettes panel to display the corresponding palette.
2. Drag a container rectangle (an `IlvGadgetContainerRectangle` for example) to the Gadgets buffer window.
3. Save the buffer as `mymainpanel.ilv`.
4. Click the Panel Class Palette icon  in the Main window toolbar to display the Panel Class palette.
5. Click the New Panel Class icon  to add the `Mymainpanel` class to the Panel Class palette.
6. From the File menu, choose New. In the submenu that appears, choose Gadgets.
A new Gadgets buffer window opens.
7. Click Gadgets in the top pane of the Palettes panel.
8. Drag an `IlvNotebook` object to the current Gadgets buffer window.
9. Save the buffer as `notebook.ilv`.
10. Click the New Panel Class icon  to add the `Notebook` panel class to the Panel Class palette.
11. Choose <Application> from the Window menu to activate the Application buffer window.

12. Drag the `Mymainpanel` class icon from the Panel Class palette to the Application buffer window.
13. Drag the `Notebook` class icon from the Panel Class palette to the container rectangle inside `Mymainpanel`.
The container rectangle is highlighted when you drop the subpanel on it.
14. Choose New from the File menu and then Gadgets in the submenu that appears.
A new Gadgets buffer window opens.
15. Drag any objects to the buffer window (text field and message labels, for example) and save it as `firstpage.ilv`.
16. Click the New Panel Class icon  to add the `Firstpage` panel class to the Panel Class palette.
17. Choose <Application> from the Window menu to activate the Application buffer window.
18. Drag the `Firstpage` class icon from the Panel Class palette and drop it onto the notebook inside the `Mymainpanel` panel.

The following dialog box appears.



This dialog box lets you add the subpanel instance as a new notebook page or replace an existing page with a new one. A sample of the notebook is displayed in the dialog box in which you can select a page to indicate where the new page should be inserted. If you

activate the “Add a new page” toggle button, the new page is inserted after the last notebook page and the selected page is ignored.

Inspecting Subpanel Instances

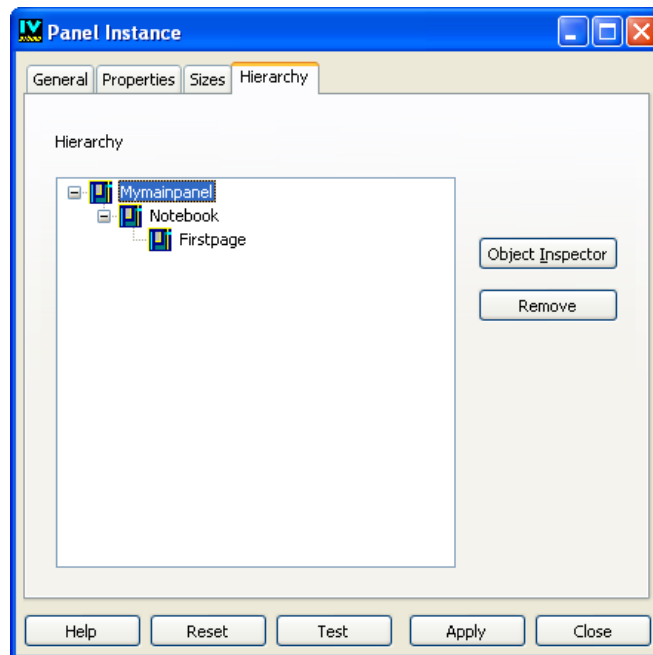
To inspect a subpanel instance:

1. Click the box in the top-left corner of `Mymainpanel` in the Application buffer window.
2. Choose Panel Inspector from the menu that appears.

The Panel Instance inspector is displayed.

3. Click the Hierarchy page.

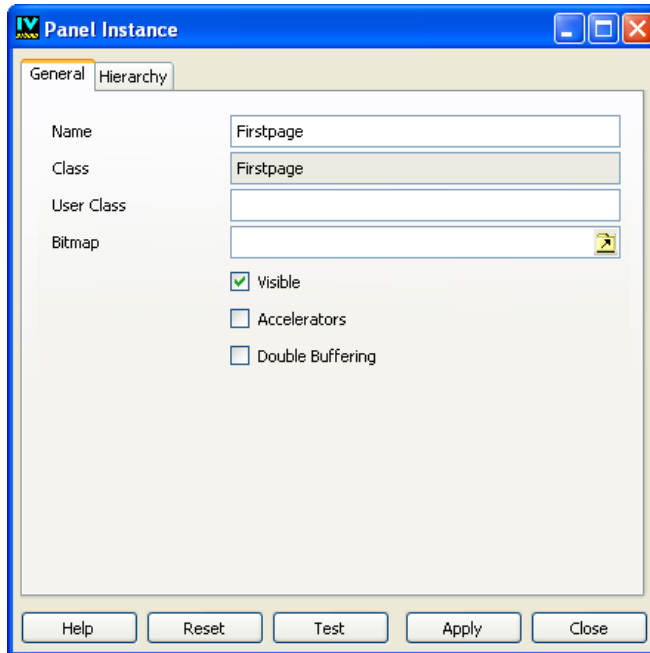
In this page, the hierarchy of the panel is displayed as a tree gadget.



You can see that the Firstpage panel is a subpanel of the Notebook panel, which is itself a subpanel of Mymainpanel.


4. Double-click the subpanel you want to inspect in the tree gadget, or select the subpanel and click Object Inspector.

The Panel Instance inspector for the subpanel appears.



5. To remove a subpanel, select it in the tree gadget in the Hierarchy page and click Remove.

Testing an Application

The Test icon  in the Main window toolbar can be used to test the application when the Application buffer is the current buffer. When you click the Test icon, a window is opened for each of the visible panel instances in the application. To close a test panel, click the Test icon again.

If you wish to test a panel instance individually, use the Test button in the Panel Instance inspector.

Using the Generated Code

This chapter uses an example to explain how to use the generated C++ code. You will find information on the following topics:

- ◆ *Building the Application*
- ◆ *Generating the C++ Code*
- ◆ *Extending the Generated Code*

Building the Application

You are going to create an application composed of the following three panels:

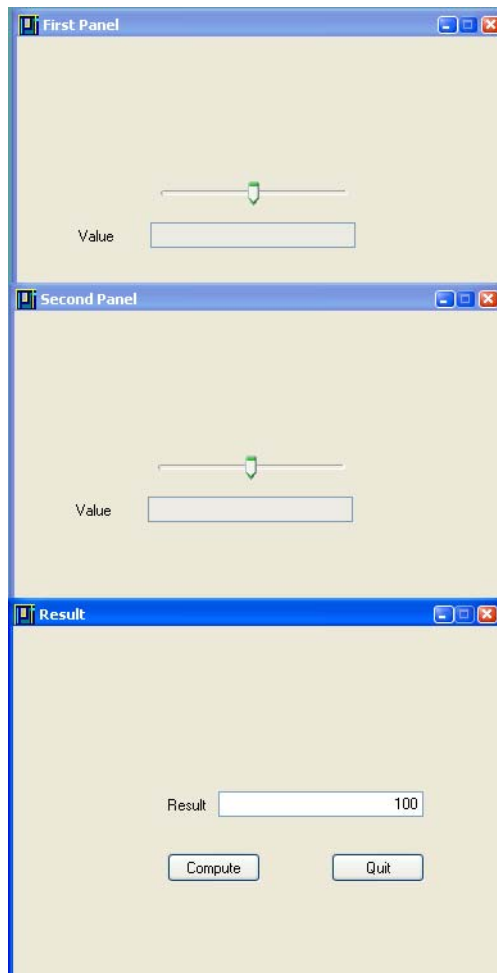


Figure 4.1 *Example Panels*

The steps to create the application are the following:

- ◆ *Setting Up the Application Class*
- ◆ *Creating the First Panel Class*
- ◆ *Creating the Second Panel Class*
- ◆ *Generating the C++ Code*

Setting Up the Application Class

The first step is to edit and save the application properties.

Let us assume that you are going to edit the default application `testapp`.

1. In the Main window, choose Application Inspector from the Application menu to open the Application inspector.
2. Change the Class name to `MyApplication`. Click Apply, then Close.
3. In the Application buffer window, choose Save As... from the File menu and save your application as `myappli.iiva` in a directory of your choice.

This operation sets the file base name and a default path for all your application files.

4. Check the application default directories by inspecting the panel again.

The header, source, and object directories default to the data directory. You can place these generated files in different directories by specifying the directories of your choice in the corresponding text fields of the Application inspector.

Creating the First Panel Class

The First and Second Panels (see Figure 4.1) are two instances of the same panel class (the class `FirstPanelClass`). This means that they have the same contents, but their names, titles, and positions are different.

You are now going to build the panel class for the two instances of this panel class.

Creating the Panel Data File

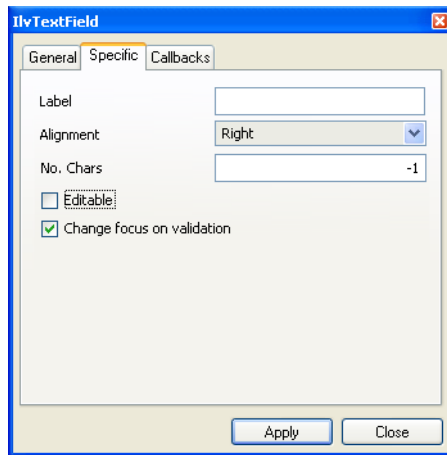
1. If necessary, open a new Gadget buffer window. In the Main window, choose New from the File menu and then Gadgets in the submenu that appears.
2. In the top pane of the Palettes panel, click Gadgets.
3. Drag the following gadgets from the bottom pane of the Palettes panel and drop them in the Gadgets buffer window:
 - Message label (class name: `IlvMessageLabel`)
 - Text field (class name: `IlvTextField`)

When a gadget is selected, its class name appears in the message area at the bottom of the Main window.

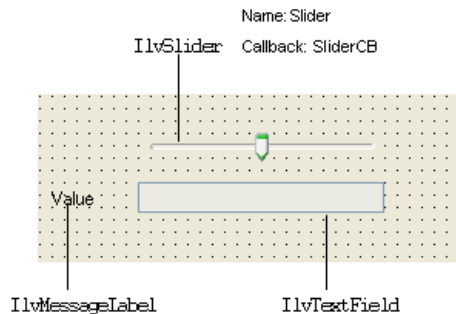
4. Double-click the message label to open its inspector panel.
5. In the Specific page of the Message Label inspector:
 - Delete the text in the Label field.
 - Type `Value` in the Label field.
6. Click Apply, then Close.

7. Double-click the text field to open its inspector panel.
8. In the Specific page of the Text Field inspector:
 - Delete the text in the Label field.
 - Select Right in the Alignment option menu.
 - Turn off the Editable toggle button.

The Text Field inspector should look like this:

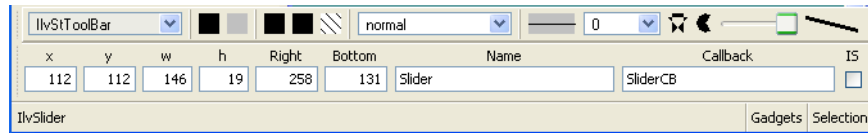


9. Click Apply, then Close.
10. In the top pane of the Palettes panel, click Miscellaneous.
11. From the bottom pane, drag the horizontal slider (class name: IlvSlider) and drop it in the Gadgets buffer window.
12. Move and resize the objects, then resize the panel so that it looks like in the figure below.



13. Select the slider.
14. Type `Slider` in the Name field of the generic inspector.
15. Type `SliderCB` in the Callback field of the generic inspector.



The generic inspector should look like this:




16. Select the text field.
17. Type `TextField` in the Name field of the generic inspector.
18. Choose `Save As...` from the File menu and save the panel as `class1.ilv` in the directory of your choice.

Setting Up the Panel Class

To set up the panel class:


1. Make sure that `class1` is the current buffer.
2. Click the Panel Class Palette icon  in the Main window tool bar to open the Panel Class palette.
3. Create the new panel class by clicking the New Panel Class icon  in the Panel Class palette.

An icon representing the new panel class, with the title `Class1`, appears in the palette.

4. Select the `Class1` panel class and click the Panel Class Inspector icon  in the Panel Class palette tool bar to open its inspector.
5. On the General page of the inspector, type `FirstPanelClass` in the Class field.
6. Click Apply, then Close.

Creating the First Panel

To create the first instance of the class `FirstPanelClass`:

1. Choose the Edit Application icon  from the tool bar in the Main window to edit the application.

The Application buffer window is activated and the Panel Class palette is displayed (if it isn't already displayed).

2. To create an instance of the first panel, drag the icon from the Panel Class palette to the Application buffer window.

An instance of `FirstPanelClass` appears in the Application buffer window.

3. Double-click the panel title bar to inspect the panel instance.

The Panel Instance inspector appears.

4. In the Panel Instance inspector, type `FirstPanel` in the Name field.

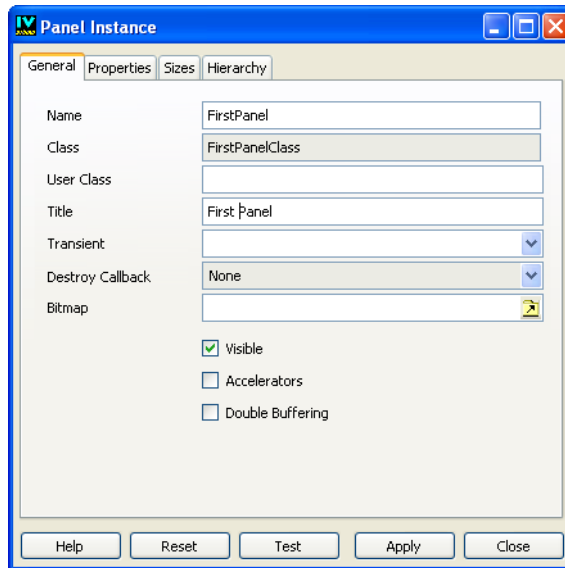
This name is used to retrieve this panel.

5. In addition, the title can be changed to `First Panel` and the position (x,y) can be moved to $(200,200)$.

Specify the title in the General page.

Specify the position of the panel in the Sizes page.

The Panel Instance inspector should look like this:



6. Click Apply to validate the `FirstPanel` options, then click Close.

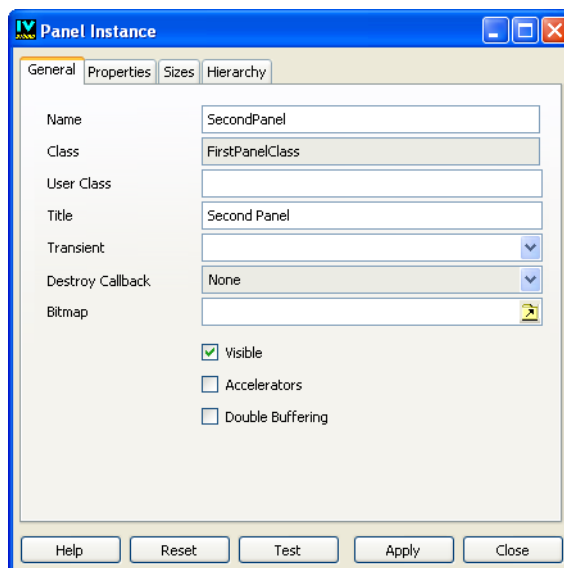
Creating the Second Panel Instance

To create a second instance of the class `FirstPanelClass`:

1. Drag the `FirstPanelClass` icon again from the Panel Class palette to the Application buffer window.

2. In the General page of the Panel Instance inspector of the second instance of `FirstPanelClass`, change the fields as follows:
 - Name: `SecondPanel`
 - Title: `Second Panel`
3. In the Sizes page of the same inspector, change the fields as follows:
 - x: 200
 - y: 400

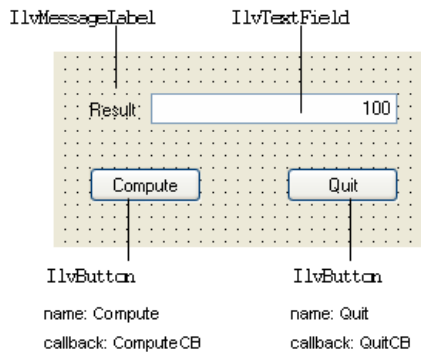
The Panel Instance inspector should look like this:



4. Click Apply, then Close.

Creating the Second Panel Class

You are now going to build the panel class for the instance of the Second Panel class illustrated below. (This is the Results Panel in Figure 4.1.)




Creating the Panel Data File

1. Open a new Gadgets buffer window. In the Main window, choose New from the File menu and then Gadgets in the submenu that appears.
2. Edit the new buffer to create the panel illustrated above.
3. Choose Save As... from the File menu to save it as `class2.ilv` in the directory of your choice.


Setting Up the Panel Class

To set up the panel class:

1. Make sure that `class2` is the current buffer.
2. Select Panel Class Palette in the Application menu to open the Panel Class palette (if it isn't already open).

3. Click the New Panel Class icon  in the Panel Class palette tool bar to create the panel class.

An icon representing the new panel class, with the title `Class2` appears in the palette.

4. Select the `class2` panel class and click the Panel Class Inspector icon  in the Panel Class palette to examine it.

The Panel inspector appears.

5. On the General page of the Panel inspector, type `SecondPanelClass` in the Class field.
6. Click Apply, then Close.

Creating the Second Panel

1. Click the Edit Application icon  in the Main window toolbar to edit the application.

The current Application buffer window is opened and the Panel Class palette is displayed (if it isn't already open).

2. To create an instance of the second panel, drag the `SecondPanelClass` icon from the Panel Class Palette to the Application buffer window.

An instance of `SecondPanelClass` appears in the Application buffer window.

3. Double-click the `SecondPanelClass` title bar.

The Panel Instance inspector appears.

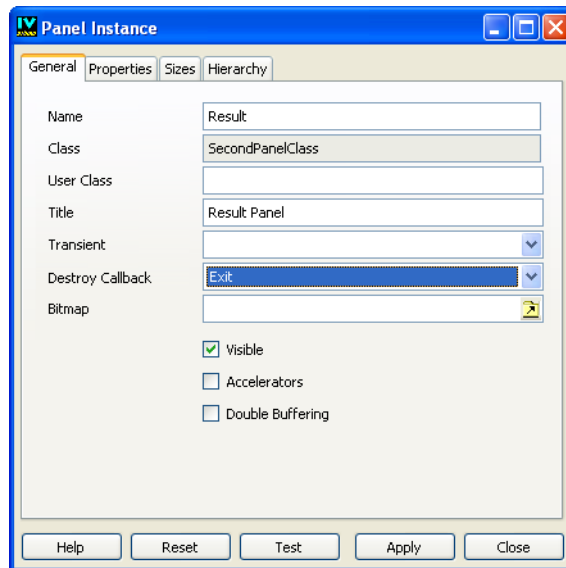
4. In the General page of the Panel Instance inspector, type `Result` in the Name field.

This name can be used to retrieve this panel.

5. Change the other fields in the appropriate inspector page as follows:

- Title: `Result Panel`
- Destroy Callback: `Exit`
- x: 200
- y: 650

The inspector should look like this:



6. Click Apply to validate the options, then click Close to close the inspector.

Generating the C++ Code

To generate the C++ code for the first time, choose Generate All from the Application menu. For our application, IBM® ILOG® Views Studio generates the following files:

- ◆ `FirstPanelClass`
 - `class1.ilv` Contains the data for the `FirstPanelClass` panels.
 - `class1.h` Header file for the `FirstPanelClass`.
 - `class1.cc` Source file for the `FirstPanelClass` (if the C++ source file extension is `.cc` on your platform).
- ◆ `SecondPanelClass`
 - `class2.ilv` Contains the data for the `SecondPanelClass` panel.
 - `class2.h` Header file for the `SecondPanelClass`.
 - `class2.cc` Source file for the `SecondPanelClass`.
- ◆ `MyApplication`
 - `myappli.iva` Contains the description of the application.
 - `myappli.h` Header file for the application class.
 - `myappli.cc` Source file for the application, which also includes the main function.
 - `myappli.mak` Simple make file for compiling and testing the application.

The following sections describe the `class1.h`, `class1.cc`, `myappli.h`, and `myappli.cc` files.

FirstPanelClass Header File

The `class1.h` header file is generated as follows:

```
// ----- *- C++ -*-----
// File: /tmp/test/class1.h
// IlogViews 4.0 generated header file
// File generated Wed May 03 16:56:53 2000
//      by IBM ILOG Views Studio
// -----
#ifndef __class1_header__
#define __class1_header__

#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/textfd.h>
#include <ilviews/gadgets/msglabel.h>
#include <ilviews/gadgets/slider.h>

// -----
class FirstPanelClass
```

```

: public IlvGadgetContainer {
public:
    FirstPanelClass(IlvDisplay* display,
                   const char* name,
                   const char* title,
                   IlvRect* size = 0,
                   IlvBoolean useAccelerators = IlvFalse,
                   IlvBoolean visible = IlvFalse,
                   IlvUInt properties = 0,
                   IlvSystemView transientFor = 0)
    : IlvGadgetContainer(display,
                          name,
                          title,
                          size ? *size : IlvRect(0, 0, 219, 58),
                          properties,
                          useAccelerators,
                          visible,
                          transientFor)
    { initialize(); }
    FirstPanelClass(IlvAbstractView* parent,
                   IlvRect* size = 0,
                   IlvBoolean useacc = IlvFalse,
                   IlvBoolean visible = IlvTrue)
    : IlvGadgetContainer(parent,
                          size ? *size : IlvRect(0, 0, 219, 58),
                          useacc,
                          visible)
    { initialize(); }
// _____
virtual void SliderCB(IlvGraphic*);
IlvSlider* getSlider() const
{ return (IlvSlider*)getObject("Slider"); }
IlvTextField* getTextField() const
{ return (IlvTextField*)getObject("TextField"); }
protected:
    void initialize();
};

#endif /* !__class1__header__ */

```

Header

The first lines of the `class1.h` panel class header file give the date and file location for the generated file. It also tells you what IBM ILOG Views version you are using.

Included Header Files

The following lines show the necessary header files for the generated class:

```

#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/textfd.h>
#include <ilviews/gadgets/msglabel.h>
#include <ilviews/gadgets/slider.h>

```

For each object contained in the generated panel, IBM ILOG Views Studio searches for its associated header file. In our example, the class `FirstPanelClass` has to include the files

<ilviews/gadgets/textfd.h>, <ilviews/gadgets/msglabel.h>, and <ilviews/gadgets/slider.h> for its text field, message label, and slider objects, respectively.

Base Class

Since we have not modified the base class name for `FirstPanelClass`, the generated class is derived from `IlvGadgetContainer`.

Constructors

Two public constructors are generated:

```
FirstPanelClass(IlvDisplay* display,
               const char* name,
               const char* title,
               IlvRect* size = 0,
               IlvBoolean useAccelerators = IlvFalse,
               IlvBoolean visible = IlvFalse,
               IlvUInt properties = 0,
               IlvSystemView transientFor = 0)
: IlvGadgetContainer(display,
                    name,
                    title,
                    size ? *size : IlvRect(0, 0, 219, 58),
                    properties,
                    useAccelerators,
                    visible,
                    transientFor)
{ initialize(); }
FirstPanelClass(IlvAbstractView* parent,
               IlvRect* size = 0,
               IlvBoolean useacc = IlvFalse,
               IlvBoolean visible = IlvTrue)
: IlvGadgetContainer(parent,
                    size ? *size : IlvRect(0, 0, 219, 58),
                    useacc,
                    visible)
{ initialize(); }
```

The first constructor builds the panel as a main window. The second builds the panel as a part of a parent view that is an `IlvAbstractView`.

Callback

Because the callback `SliderCB` is assigned to the slider, IBM ILOG Views Studio generates its related virtual member function:

```
virtual void SliderCB(IlvGraphic*);
```

Named Objects

Our two named objects, `Slider` and `TextField`, can be accessed by the following generated member functions:

```
IlvSlider* getSlider() const
{ return (IlvSlider*)getObject("Slider"); }
IlvTextField* getTextField() const
```

```
{ return (IlvTextField*)getObject("TextField"); }
```

If you do not want IBM ILOG Views Studio to generate these functions, turn off the Names toggle button in the Panel Class inspector (Options notebook page).

FirstPanelClass Source File

Header

The `FirstPanelClass` source file starts with following header lines:

```
// ----- *- C++ *-
// File: /tmp/test/class1.cc
// IlogViews 4.0 generated source file
// File generated Wed May 03 16:56:53 2000
//     by IBM ILOG Views Studio
// -----
```

Class Header File

The panel class header file is the first included file:

```
#include <class1.h>
```

Panel Data

The lines between the `#include` statement and the callback definition `_SliderCB` define the way the panel data is loaded when the panel class constructor is called. If the Data toggle button of the Panel Class inspector (Options notebook page) is turned on, the panel data is generated in a constant character string. In this case, instead of loading the data from a file, the panel can load its description from the generated string (through an `istream`), unless there is a compiler limitation.

Callback

Because the callback name `SliderCB` is assigned to a panel object, IBM ILOG Views Studio generates the following callback:

```
static void ILVCALLBACK
_SliderCB(IlvGraphic* g, IlvAny)
{
    FirstPanelClass* o = (FirstPanelClass*)
        g->GetCurrentCallbackHolder()->getContainer();
    if (o) o->SliderCB(g);
}
```

This function gets the panel class from the graphic object and calls the related method that is declared in the class declaration.

Callback Method Definition

Because the Callbacks Definitions toggle button of the Panel Class inspector (Options notebook page) is turned on, the default definition of the callback method `SliderCB` is

generated in the source file. This lets you compile, link and test your application before writing the real function definition.

The generated callback method looks like this:

```
void
FirstPanelClass::SliderCB(IlvGraphic* g)
{
    const char* className = g->className();
    IlvPrint(" %s : SliderCB method ...",className);
}
```

When called by a slider, this function prints the following message:

```
IlvSlider : SliderCB method ...
```

If you do not want IBM ILOG Views Studio to generate this callback method definition, turn off the Callbacks Definitions toggle button of the Panel Class inspector (Options notebook page). In this case, you must write your own version of `FirstPanelClass::SliderCB` in a separate file and link that file object to the application.

initialize Member Function

The generated constructors of the class `FirstPanelClass` call the `initialize` member function to initialize the panel.

The `initialize` function loads the panel contents from a file or an `istrstream`, according to the platform you use to compile the application.

```
void
FirstPanelClass::initialize()
{
#ifdef ILVNOSTATICDATA
    readFile(FILENAME);
#else /* !ILVNOSTATICDATA */
    istrstream str((char*)_data);
    read(str);
#endif /* !ILVNOSTATICDATA */
    registerCallback("SliderCB", _SliderCB);
}
```

MyApplication Header File

Header

Like all the generated files, the first lines of the application header file, `myappli.h`, show the date and directory path for the generated file, as well as providing the version of IBM® ILOG® Views:

```
// ----- *- C++ *-
// File: /tmp/test/myappli.h
// IlogViews 4.0 generated application header file
// File generated Wed May 03 16:56:53 2000
//     by IBM ILOG Views Studio
// -----
```

Included Header Files

The application header file includes the default base class header file and its panel classes header files:

```
#include <ilviews/gadgets/appli.h>
#include <class1.h>
#include <class2.h>
```

MyApplication Class

IBM ILOG Views Studio generates the following application class:

```
class MyApplication: public IlvApplication {
public:
    MyApplication(
        const char* appName,
        const char* displayName = 0,
        int argc = 0,
        char** argv = 0
    );
    MyApplication(
        IlvDisplay* display,
        const char* appName
    );
    ~MyApplication();
    virtual void makePanels();
    virtual void beforeRunning();
    FirstPanelClass* getFirstPanelClass() const
        { return (FirstPanelClass*) getPanel("FirstPanelClass"); }
    FirstPanelClass* getSecondPanel() const
        { return (FirstPanelClass*) getPanel("SecondPanel"); }
    SecondPanelClass* getResult() const
        { return (SecondPanelClass*) getPanel("Result"); }
};
```

Base Class

The generated class is derived from `IlvApplication`. See `IlvApplication` in the IBM ILOG Views *Reference Manual* for a description of this class, which is part of the IBM ILOG Views library.

Constructors

The generated constructors only call the related base class constructors.

makePanels Method

For each application, IBM ILOG Views Studio generates the `makePanels` method that is called when the application is initialized. This method handles the creation of the generated application panels. See the class `IlvApplication` in the IBM ILOG Views *Reference Manual*.

Panel Accessors

For each panel in the application IBM ILOG Views Studio generates a panel accessor which returns the panel.

MyApplication Source File

Header

As usual, the first lines of the application source file, `myappli.cc`, show file, date and path directory for the generated file, as well as providing the version of IBM® ILOG® Views:

```
// ----- *- C++ -*-----  
// File: /tmp/test/myappli.cc  
// IlogViews 4.0 generated application source file  
// File generated Wed May 03 16:56:53 2000  
//      by IBM ILOG Views Studio  
// -----
```

Class Header File

The application source file always includes the generated application class header file:

```
#include <myappli.h>
```

makePanels Function Definition

The generated `makePanels` member function looks like this:

```
void  
MyApplication::makePanels()  
{  
    // --- parameters ---  
    IlvDisplay*      display = getDisplay();  
    IlvRect          bbox;  
    IlvContainer*   cont;  
    // --- FirstPanel ---  
    bbox.moveResize(200, 200, 500, 500);  
    cont = new FirstPanelClass(display,  
                                "FirstPanel",  
                                "First Panel",  
                                &bbox,  
                                IlFalse,  
                                IlFalse, 0, 0);  
  
    addPanel(cont);  
    cont->show();  
    // --- SecondPanel ---  
    bbox.moveResize(200, 300, 500, 500);  
    cont = new FirstPanelClass(display,  
                                "SecondPanel",  
                                "Second Panel",  
                                &bbox,  
                                IlFalse,  
                                IlFalse, 0, 0);  
  
    addPanel(cont);  
    cont->show();  
    // --- Result ---
```

```

bbox.moveResize(200, 400, 500, 500);
cont = new SecondPanelClass(display,
                            "Result",
                            "Result Panel",
                            &bbox
                            IlFalse,
                            IlFalse, 0, 0);

addPanel(cont);
cont->setDestroyCallback(IlvAppExit, this);
cont->show();
// --- The Exit panel is not wanted ---
setUsingExitPanel(IlFalse);
}

```

This application contains three panels: `FirstPanel` and `SecondPanel` are part of the class `FirstPanelClass`. The following points should be noted:

- ◆ Each panel is created at the position specified in the x and y fields of the Panel Instance inspector (Sizes notebook page).
- ◆ The size of the rectangle passed to the panel constructor does not really affect the panel sizes, since they are resized when their data is loaded. If the Generate Size toggle button of the Panel Instance inspector (Sizes notebook page) is turned on, the Bounding Box Width and Height values specified in the Panel Instance inspector are used to resize the panel after it is created.
- ◆ Each panel is added to the application after being created:

```
addPanel(cont);
```

- ◆ If the Visible toggle button in the Panel Instance inspector (General notebook page) is turned on, that panel is shown by the `show()` member function:

```
cont->show();
```

- ◆ The following code is generated because the destroy callback of the Result panel is set to Exit in the Panel Instance inspector (General notebook page):

```
cont->setDestroyCallback(IlvAppExit, this);
```

- ◆ Since the Exit panel is not wanted, the following code is generated:

```
setUsingExitPanel(IlFalse);
```

main Function

Because the `main()` toggle button is checked in the Options notebook page of the Application Inspector, the `main` function is generated in the application source file:

```

main(int argc, char* argv[])
{
    // IlvSetCurrentCharSet(<YourCharSet>);
    IlvSetLanguage();
    MyApplication* appli = new MyApplication("myappli", 0, argc,
    argv);
}

```



```

    if (!appli->getDisplay())
        return -1;
    appli->run();
    return 0;
}

```

This function creates an application of class `MyApplication`. Before running the application, the function checks whether the created application succeeded in creating a display.

If you do not want the `main` function to be generated, turn off the `main()` toggle button.

Testing the Generated Application

To test the generated application, run the `make` utility in the application object directory using the generated make file, then launch `myappli`. Following is an example of the commands. If the object directory is `/tmp/test`:

```

cd /tmp/test
make -f myappli.mak
myappli

```

To end the application, close the Result panel with your window manager.

Extending the Generated Code

To add a member function and write an appropriate version of `SliderCB`, you will derive the `MyFirstPanelClass` from `FirstPanelClass`.

Defining a Derived Class

In the file `myclass1.h`, you will declare `MyFirstPanelClass` like this:

```

#include <class1.h>

class MyFirstPanelClass
: public FirstPanelClass {
public:
    MyFirstPanelClass(IlvDisplay* display,
                     const char* name,
                     const char* title,
                     IlvRect* size = 0,
                     IlvBoolean useAccelerators = IlvFalse,
                     IlvBoolean visible = IlvFalse,
                     IlvUInt properties = 0,
                     IlvSystemView transientFor = 0):
        FirstPanelClass(display,
                        name,
                        title,
                        size,

```

```

        useAccelerators,
        visible)
    {}
    virtual void SliderCB(IlvGraphic*);
    IlvInt getValue() const { return getTextField()->getIntValue(); }
};

```

Base Class

`MyFirstPanelClass` is derived from `FirstPanelClass`.

Constructor

You only need to define one constructor. This constructor builds the panel as a main window and calls its base class constructor.

Callback Method

The `SliderCB` virtual member function is redefined in the derived class to display the slider value in the text field. Here is a possible definition of such a function:

```

void
MyFirstPanelClass::SliderCB(IlvGraphic*)
{
    getTextField()->setValue(getSlider()->getValue(), IlTrue);
}

```

getValue Member Function

To get the value displayed by the panel, you must define this inline member function:

```

IlvInt getValue() const { return getTextField()->getIntValue(); }

```

Using the Derived Class

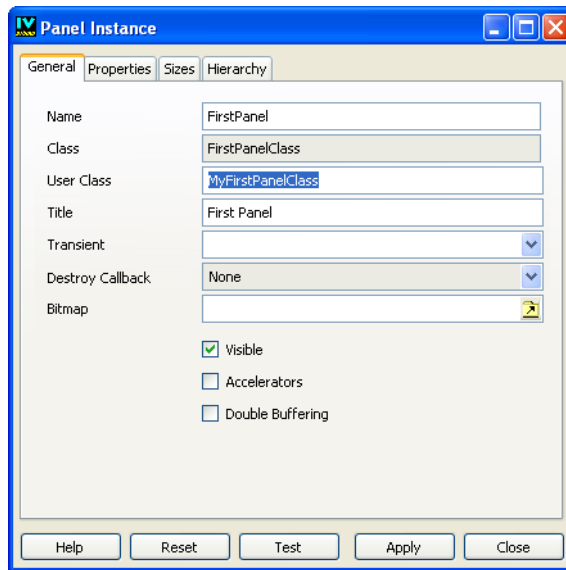
To use `MyFirstPanelClass` instead of `FirstPanelClass`, which is needed to create `FirstPanel`:

- ◆ Set the User Class field in the Panel Instance inspector.
- ◆ Insert a `#include` statement in the generated application header file.

Setting Up the User Class

To set up the user class:

1. In the Application buffer window, double-click the first instance of `FirstPanelClass`.
2. In the Panel Instance inspector, type `MyFirstPanelClass` in the User Class field:



3. Click Apply, then Close.

Note: Steps 2 and 3 can be repeated for SecondPanel.

Instead of generating the following code in the `makePanels` function:

```
cont = new FirstPanelClass(display,
                           "FirstPanel",
                           "First Panel",
                           &bbox,
                           IlFalse,
                           IlFalse, 0, 0);
);
```

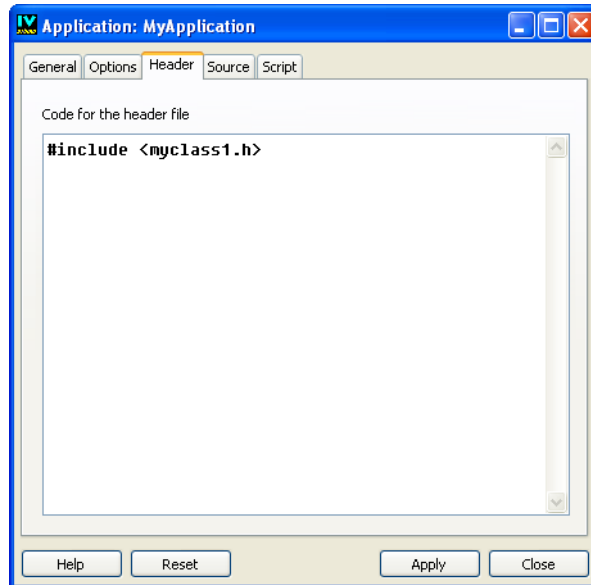
IBM ILOG Views Studio generates the code below in the application source file `myappli.cc`:

```
cont = new MyFirstPanelClass(display,
                              "FirstPanel",
                              "First Panel",
                              &bbox,
                              IlFalse,
                              IlFalse, 0, 0);
);
```

Since `MyFirstPanelClass` is declared in `myclass1.h`, you need to include this file in `myappli.h`.

Inserting Code in the Generated Application Header File

1. Choose Application Inspector from the Code menu. The Application inspector is displayed.
2. In the Application inspector, open the Header notebook page.
3. In the Header notebook page, type `#include <myclass1.h>`:



4. Click Apply, then Close.

When generating the application source file again, IBM ILOG Views Studio inserts the following expression in the application header file:

```
// -----
// --- Inserted code

#include <myclass1.h>

// --- End of Inserted code
```

Linking Additional Object Files

The make file generated by IBM ILOG Views Studio takes care of compiling the generated files and linking the application. In addition, it can link your own object files to the application through the make `USEROBJ`s variable. You are responsible for your own object files. However, you can write your own make file to maintain the additional object files by copying the make options generated by IBM ILOG Views Studio.

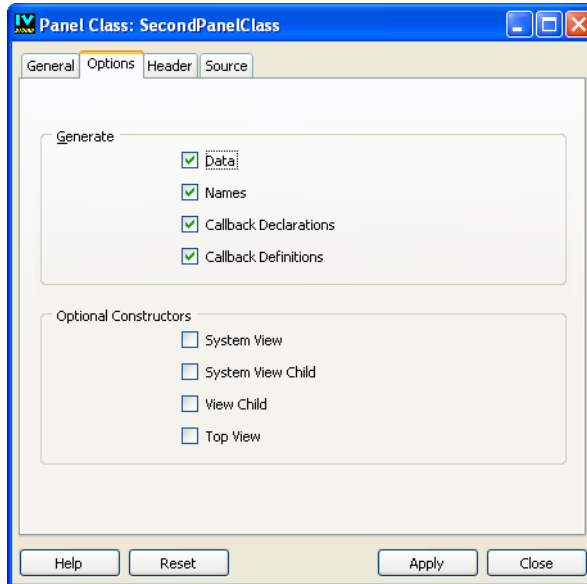
For example, if the definition of the `MyFirstPanelClass::SliderCB` is in your object file `myclass1.o`, you can use the generated make file `myappli.mak` like this:

```
make -f myappli.mak USEROBSJ=myclass1.o
```

Defining Callbacks without Deriving Classes

In the previous section, we derived a panel class and defined the callback methods in the derived class. For `SecondPanelClass`, we will now insert its callback methods in the generated source file without deriving a subclass:

1. In the Panel Class palette, select `SecondPanelClass` and click the Panel Class Inspector icon in the Panel Class palette tool bar.
2. In the Panel Class inspector, go to the Options notebook page and turn off the Callback Definitions toggle button.



3. Open the Source notebook page and type the following code in the section "Code for the source file":

```
#include <ilviews/gadgets/appli.h>
#include <myclass1.h>

void
SecondPanelClass::ComputeCB(IlvGraphic*)
{
    IlvApplication* appli = IlvApplication::GetApplication(this);
    MyFirstPanelClass* pan1 =
        (MyFirstPanelClass*)appli->getPanel("FirstPanel");
```

```

MyFirstPanelClass* pan2 =
    (MyFirstPanelClass*) appli->getPanel("SecondPanel");
getResult()->setValue(pan1->getValue() + pan2->getValue(), IlTrue);
}
void
SecondPanelClass::QuitCB(IlvGraphic*)
{
    delete IlvApplication::GetApplication(this);
    IlvExit(0);
}

```

4. Click Apply to validate the operation, and Close to quit the inspector.
5. In the Panel Class User Code panel, click Apply.

When generating again, IBM ILOG Views Studio inserts your two callback methods in the `class2.cc` file.

Customizing the Gadgets Extension of IBM ILOG Views Studio

This chapter provides a list of the configuration options for the Gadgets extension of IBM® ILOG® Views Studio. You can use these options to customize Studio.

- ◆ *Configuration Options for the Gadgets Extension*

Configuration Options for the Gadgets Extension

IBM® ILOG® Views Studio provides you with the following configuration options for the Gadgets extension:

- ◆ `additionalLibraries "<library list>"` lets you specify a list of IBM® ILOG® Views libraries to link into the generated application.

For example:

```
studio {  
    additionalLibraries "ilvadvgadmgr ilvgadmgr ilvmgr";  
}
```

- ◆ `applicationBaseClass <className>` lets you specify the name of the base class of the generated application class. The default value is `IlvApplication`.

- ◆ `applicationBufferBackground` "`<colorName>`" lets you specify the background color of the Application buffer window. The default value is "Cadet Blue".
- ◆ `applicationFileExtension` "`<extension>`" lets you specify the extension of the application file. The default value is ".iva".
- ◆ `applicationHeaderFile` "`<header>`" lets you specify the header file to be included in the generated application header file. The default value is "`<ilviews/appli.h>`".
- ◆ `defaultApplicationName` `<name>` lets you specify the name of a new application. The default value is `testapp`.

For example:

```
studio {
    defaultApplicationName newappli;
}
```

- ◆ `defaultCallbackLanguage` `<language>` lets you specify the callback language used by default when a callback is attached to an object. This option only applies to IBM ILOG Views Studio Script extension (`jsstudio`). Its default value is `JvScript`. If you do not want callbacks written in `jsstudio` to default to `JvScript`, set this option to `none`.

For example:

```
studio {
    defaultCallbackLanguage none;
}
```

- ◆ `defaultHeaderDir` "`<dir>`" lets you specify the header file directory set by default for new applications. The specified directory is relative to the application directory. Once the application is created, this directory can be modified via the Application inspector.

For example:

```
studio {
    defaultHeaderDir "include";
}
```

- ◆ `defaultHeaderFileScope` "`<dir>`" lets you specify a subdirectory which is generated in the `#include` statements. Once the application is created, the header file scope can be modified via the Application inspector.

For example:

```
studio {
    defaultHeaderFileScope "myinclude/";
}
```

- ◆ `defaultObjDir` "`<dir>`" lets you specify the makefile directory that is set by default for new applications. The specified directory is relative to the application directory. Once the application is created, it can be modified through the Application Inspector.

For example:

```
studio {
    defaultObjDir "obj";
}
```

- ◆ `defaultSrcDir "<dir>"` lets you specify the source file directory that is set by default for new applications. The specified directory is relative to the application directory. Once the application is created, this directory can then be modified through the Application Inspector.

For example:

```
studio {
    defaultObjDir "obj";
}
```

- ◆ `defaultSystemName "<name>"` lets you specify the target system for which you want to generate the application makefile. This information is not specific to an application. It concerns only the system you use to compile the generated application. By default, the makefile is generated for the system on which IBM ILOG Views Studio is running. Use this option if you want to modify the platform for which the makefile will be generated by default.

For example:

```
studio {
    defaultSystemName "sparc_5_4.0";
}
```

- ◆ `headerFileExtension "<extension>"` lets you specify the extension of the generated header file. The default value is ".h".

For example:

```
studio {
    headerFileExtension ".hxx";
}
```

- ◆ `JvScriptApplication <true/false>` lets you specify whether the generated C++ application will use IBM ILOG Script for IBM ILOG Views. This option is only applicable when you use the GUI Application plug-in with the `jsstudio` extension.

For example:

```
studio {
    JvScriptApplication false;
}
```

- ◆ `makeFileExtension "<extension>"` lets you specify the extension of the generated makefile. The default value is ".mak". For example:

```
studio {
    makeFileExtension ".mk";
}
```

- ◆ `noPanelContents <true/false>` lets you specify whether the contents of the panel instances must be loaded when you open an application file. This option defaults to `false`. Use this option to reduce the loading time if you often edit applications containing a lot of panels. Then, you can explicitly load the contents of a panel by choosing Load Contents from the panel instance menu in the Application buffer window.
- ◆ `panelBaseClass <className>` lets you specify the base class name that will be automatically given to newly created panel classes, whatever the buffer type. If this option is not specified, the base class name will depend on your buffer type.

For example:

```
studio {
    panelBaseClass MyGadgetContainer;
}
```

- ◆ `sourceFileExtension "<extension>"` lets you specify the extension of the generated C++ source file regardless of the selected target platform. The default value depends on the target platform that is selected in the Application inspector panel.
- ◆ `system <systemDescription>` declares the information related to the target platform, needed by the editor to generate your application files. This option can be repeated. Its format is the following:

```
system "<system-name>" {
    <option-1> <value-1>;
    ...
    <option-n> <value-n>;
}
```

`system-name` is the IBM ILOG Views platform name, such as `msvc5` or `sparc_5_4.0`. You should not have to modify these options since they are given for all the platforms on which IBM ILOG Views is available. Following is the list of the possible options used in the system description:

- `compiler "<command>"` specifies the command to run the compiler on this platform.
- `compilerOptions "<options>"` specifies the options to be passed to the compiler for producing an object file.
- `linker "<command>"` specifies the command to run the linker on this platform.
- `linkerOptions "<options>"` specifies the options to be passed to the linker for producing an executable file.
- `libraries "<libraries>"` lists the IBM ILOG Views libraries to link with for producing an executable file.
- `systemLibraries "<libraries>"` lists the system libraries to link with for producing an executable file.

- `motif` Either `true` or `false` to indicate whether the platform can use Motif.
- ◆ `toolBarItem <commandName> <toolBarName> [-before <refCommandName>]` lets you add a command `<commandName>` in the tool bar `<toolBarName>`. This option can be repeated. Optionally, you can specify a command `<refCommandName>` before which you want to insert the new command by using the keyword `-before`.

For example:

```
studio {  
    toolBarItem SelectLabelMode IlvStGadgetBuffer -before  
    SelectFocusMode;  
}
```

- ◆ `userSubClassPrefix "<prefix>"` lets you customize the prefix of the class name for a generated panel subclass. By default, this prefix is "My".
- ◆ `userSubClassSuffix "<suffix>"` lets you specify the suffix of the class name for a generated panel subclass.

Extending IBM ILOG Views Studio

This chapter describes additional ways to extend IBM® ILOG® Views Studio when you have installed the Gadgets extension. It contains the following sections:

- ◆ *Extending IBM ILOG Views Studio Components*
- ◆ *Integrating your Own Graphic Objects*
- ◆ *Extending IBM ILOG Views Studio: An Example*

Note: To extend IBM ILOG Views Studio, you need the IBM ILOG Views Gadgets, and IBM ILOG Views Manager packages.

Extending IBM ILOG Views Studio Components

This section describes the IBM® ILOG® Views Studio components that you can extend.

- ◆ *Defining a New Command*
- ◆ *Defining a New Panel*
- ◆ *IBM ILOG Views Studio Messages*
- ◆ *Defining a New Buffer*

- ◆ *Defining a New Editing Mode*
- ◆ *The Class `IlvStExtension`*

Defining a New Command

A command is an action that the user can perform using the editor. A command is a C++ class, `IlvStCommand`, which is described in the file `<ivstudio/command.h>`. It is defined by:

- ◆ A declaration that contains information to be displayed in a menu, an icon, or a help message and a list of message names to be sent when the command has been successfully executed. Predefined commands are declared in the command description file `studio.cmd`. For more information on this file, see the section “IBM ILOG Views Studio Command File” in the IBM ILOG Views *Studio User’s Manual*.
- ◆ An action that is defined in the virtual member function `doIt`. It returns 0 when no error occurs; otherwise it returns the corresponding error.

Command Errors

An IBM ILOG Views Studio error is a subclass of the `IlvStError` class, which is declared in the file `<ivstudio/error.h>`. An error can be returned by the `doIt` member function of a command. An error is defined by a string message and a type. There are three types of errors:

- ◆ `IlvStInformation`
- ◆ `IlvStWarning`
- ◆ `IlvStFatal`

To add a new command, do the following:

1. Define a subclass of `IlvStCommand` to define the virtual member function `doIt`.
2. Add a descriptor in a command declaration file or directly in an option file.
3. If you use a new command declaration file, declare it in your option file using the `commandFile` option.
4. Register the command in the editor using the member function `IlvStudio::registerCommand`, giving the command name and a function to build an instance of the command.

Predefined Command Classes

There are two subclasses for common needs:

- ◆ `IlvStClickAddObject` to add a new object to the current buffer.

- ◆ `IlvStShowPanel` to display a panel on the screen. The construction of the instance uses the panel to be displayed. If the panel is already visible, this command just hides the panel.

Executing a Command

If the editor is requested to execute a command and it fails, the corresponding error is returned by the command execution procedure and is managed by the editor. If it succeeds, the list of messages associated with the command are sent by the editor. It means that the subscriptions attached to each message are executed.

Defining a New Panel

The IBM® ILOG® Views Studio interface is composed of several panels. Panels are instances of a subclass of `IlvStPanelHandler`, which is described in the file `<ivstudio/panel.h>`. This class is not a gadget container class, but rather a handle to the actual graphic panels that are instances of `IlvGadgetContainer`. It allows you to keep the graphic aspect of the panel completely separate from its behavior within IBM ILOG Views Studio. Following are the virtual member functions that may be redefined:

- ◆ `connect` initializes the panel. This method is usually called after the panel has been created. It is meant to separate the constructor from initialization.
- ◆ `apply` is associated with the `apply` callback that you can attach to any object.
- ◆ `cancel` is associated with the `cancel` callback that you can attach to any object.
- ◆ `reset` is associated with the `reset` callback that you can attach to any object.

The `show` and `hide` methods of the panel handler must be used to show and hide an IBM ILOG Views Studio panel. Never directly show or hide the handled gadget container.

The subclass `IlvStDialog` is a handle for an instance of `IlvDialog`.

IBM ILOG Views Studio Messages

An IBM® ILOG® Views Studio message contains information that describes an event that took place. A message collects subscriptions. A subscription is an action that is performed whenever a message is sent. Messages are never created by the user, but are accessed through the editor using their names.

A subscription is a subclass of the `IlvStSubscription` class, which is declared in the file `<ivstudio/message.h>`. It is associated with a receiver and has a `doIt` virtual member function. For example, when a panel wants to react to the `ObjectSelected` message that is generated each time the object selection changes in the current buffer, it subscribes to this message using a subscription instance. This can be done in the panel constructor by calling the member function `subscribe` on the message instance. The message instance is given by

the editor through its name. Then, whenever the message `ObjectSelected` is sent, the `doIt` member function of this subscription is invoked.

Defining a New Buffer

A buffer is a document that is edited in IBM® ILOG® Views Studio. It uses an `IlvManager` to display, edit, save, and read its contents. If you need to subclass the manager to save more information concerning your objects, for example, you have to subclass a corresponding buffer. The `IlvStBuffer` class is defined to encapsulate the `IlvManager`, and the `IlvStGadgetBuffer` class is defined to encapsulate an `IlvGadgetManager`. These classes are declared in the file `<ivstudio/stbuffer.h>` and `<ivstudio/gadgets/gadbuf.h>`. If you need to define a specialized manager class to edit and save your graphic objects, you have to define a corresponding buffer class.

Registering Buffer Types

When loading a `.ilv` file, IBM ILOG Views Studio first reads the file *creator class* information to determine the type of the buffer that must be created for editing this file. For example, when reading a file saved by an `IlvGadgetManager`, IBM ILOG Views Studio sees that the creator class of that `.ilv` file is `IlvGadgetManagerOutputFile` and then creates an `IlvStGadgetBuffer`. This is made possible by the `IlvStBuffers::registerType` function that allows you to associate a buffer constructor function with a file creator class. Use this function to register your own buffer types. IBM ILOG Views Studio uses an `IlvStBuffers` object to manage all the buffers. You can obtain a reference to this object by calling the `IlvStudio::buffers` function:

```
static IlvStBuffer*
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
{
    // MyGadgetBuffer is a subtype of IlvStBuffer.
    return new MyGadgetBuffer(editor, name);
}

...
editor->buffers().registerType("MyGadgetManagerOutput",
                             MakeMyBuffer);
...
```

Panel Classes

An `IlvStPanelClass` object is an IBM ILOG Views Studio object that describes the C++ panel class you wish to generate for a buffer. It contains all the information that IBM ILOG Views Studio requires to generate a subclass of `IlvContainer` using the data edited in your buffer.

An `IlvStPanelClass` object contains the class name, the base class, the base name of the file, the directories where the files are generated, and so on. Some of its properties are related to the type of the corresponding buffer, for example, the base class: a `Gadgets` buffer

(`IlvStGadgetBuffer`) is used to generate a subclass of `IlvGadgetContainer`, while a 2D buffer (`IlvStBuffer`) is used to generate a subclass of `IlvContainer`.

Let us suppose that you have defined the class `MyContainer`, a subclass of `IlvGadgetContainer` that can read additional information saved by your manager. You will then want the generated class to derive from `MyContainer`. You may specify the base class in the Panel Class inspector each time you create a panel class using your buffer, but the best way is to automatically set up the panel class so its base class defaults to `MyContainer`.

You can do this by defining the `setUpPanelClass` virtual member function for your buffer. This function is called when IBM ILOG Views Studio creates a panel class from your buffer.

```
void
MyBuffer::setUpPanelClass (IlvStPanelClass* pclass) const
{
    IlvStGadgetBuffer::setUpPanelClass (pclass);
    pclass->setBaseClass ("MyContainer");
}
```

Integrating Customized Container Classes

In many situations, IBM ILOG Views Studio creates instances of containers. For example, when you test a panel or add it to the Application buffer window. To select the appropriate classes IBM ILOG Views Studio uses a set of container information objects. An `IlvStContainerInfo` object provides the information that IBM ILOG Views Studio needs about a container subclass, and creates instances of that subclass.

To integrate a class of containers, you have to define a subclass of `IlvStContainerInfo` and add an instance of this class to the IBM ILOG Views Studio container information set as follows:

```
studio->addContainerInfo (myContainerInfo);
```

Defining a New Editing Mode

An editing mode is an IBM® ILOG® Views Studio object that encapsulates an object of type `IlvManagerViewInteractor`. To add a new editing mode:

1. Create an object of the class `IlvStMode`.
2. Add this object to the IBM ILOG Views Studio mode delegate `IlvStModes`.
3. Define a command constructor that returns an instance of the `IlvStSetMode` class.
4. Register this command constructor.
5. Declare the command descriptor in a command declaration file.

Once the editing mode has been created, you can associate it with a bitmap and add it to the tool bar to the left of the Main window.

Adding an Object to the IBM ILOG Views Studio Mode Delegate

`IlvStudio` has several “delegates” that are dedicated to handling specific services. `IlvStudio` has a member of the class `IlvStModes` to manage the modes. Its reference can be accessed by:

```
IlvStModes& IlvStudio::modes();
```

You can use the following function to add your editing mode:

```
void IlvStModes::add(IlvStMode* mode)
```

Defining a Command Constructor that Returns an Instance of the `IlvStSet-Mode Class`

The command constructor to be defined can be a simple function. The Menu mode, for example, can be coded as follows:

- ◆ Adding the new mode:

```
editor->modes().add(new IlvStMode(editor,
                                "Menu",
                                "SelectMenuMode",
                                new IlvMakeMBLinkInteractor));
```

- ◆ Command constructor function:

```
static IlvStCommand*
MkSelectMenuMode(IlvStudio*)
{
    return new IlvStSetMode("Menu");
}
```

The Class `IlvStExtension`

To initialize a new extension and add it to IBM® ILOG® Views Studio, you have to derive a class from `IlvStExtension` defining a set of methods that will be invoked in a predefined sequence. The constructor of `IlvStExtension` takes the following two parameters:

- ◆ `name` is the name of the extension.
- ◆ `editor` is the instance of the editor that is being extended.

The constructor of the `IlvStExtension` class adds the new instance to the extension list of the editor. You must create an instance of your extension before initializing the `IlvStudio` instance. When the editor is deleted, this instance is also deleted. An extension must not be explicitly deleted.

Here is an example:

```
#include <ivstudio/studext.h>

class MyStudioExtension
: public IlvStExtension {
```

```

public:
    MyStudioExtension(IlvStudio* editor);
    virtual IlvBoolean preInitialize();
    virtual IlvBoolean initializePanels();
    virtual IlvBoolean initializeCommandDescriptors();
    virtual IlvBoolean initializeBuffers();
    virtual IlvBoolean initializeInspectors();
};

int
main(int argc, char* argv[])
{
    IlvSetLanguage();
    // --- Display ---
    IlvDisplay* display = new IlvDisplay("ivstudio", "", argc, argv);
    if (display->isBad()) {
        IlvFatalError("Couldn't open display");
        delete display;
        return 1;
    }

    // ---- Create and initialize the editor ---
    IlvStudio* editor = new IlvStudio(display, argc, argv);
    if (editor->isBad()) {
        IlvFatalError("Could not initialize the editor");
        delete display;
        return 2;
    }
    new MyStudioExtension(editor); // added line
    editor->initialize();
    editor->parseArguments();
    editor->mainLoop();
    return 0;
}

```

First Initialization Step

The `preInitialize` method is the first one to be invoked when the editor is initialized. At this stage, configuration files have not yet been read.

What you should do in this method:

- ◆ Complete the display path so that it contains the directories where your configuration and data files are located.
- ◆ Add a configuration file.
- ◆ If you define a buffer type and want it to be the default buffer when the editor is initialized, you have to set the default constructor in this method.

For example:

```

// A buffer constructor.
static IlvStBuffer* ILVCALLBACK
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
{
    return new MyGadgetBuffer(editor, name);
}

```

```

static const char* UserData = "../data";

IlBoolean
MyStudioExtension::preInitialize()
{
    IlvStudio* editor = getEditor();
    // Add the path.
    editor->getDisplay()->prependToPath(UserData);
    // Add an option file.
    editor->addOptionFile("mystudio.opt");
    // Must be done here so
    // the first default buffer will be a MyGadgetBuffer.
    editor->buffers().setDefaultConstructor(MakeMyBuffer);
    return IlTrue;
}

```

Initializing Buffers

The `initializeBuffers` method is called after the predefined buffers are initialized. You can complete the buffer initialization and the related initializations in this method.

For example:

```

IlBoolean
MyStudioExtension::initializeBuffers()
{
    IlvStudio* editor = getEditor();
    editor->buffers().registerType("MyGadgetManagerOutput", MakeMyBuffer);
    return IlTrue;
}

```

Initializing Command Descriptors

The `initializeCommandDescriptors` method is called after the command descriptors are read and after the predefined command constructors are registered. You can register your command constructors in this method.

For example:

```

static IlvStCommand*
MkMyShowPanel(IlvStudio* editor)
{
    return new IlvStShowPanel(editor->getPanel("MyPanel"));
}

static IlvStCommand*
MkMyAddClass(IlvStudio*)
{
    return new MyAddClass;
}

static IlvStCommand*
MkMyNewBuffer(IlvStudio*)
{
    return new MyNewBuffer;
}

```

```

IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    // Register my commands.
    IlvStudio* editor = getEditor();
    editor->registerCommand("MyShowPanel", MkMyShowPanel);
    editor->registerCommand("AddMyClass", MkMyAddClass);
    editor->registerCommand("MyNewBuffer", MkMyNewBuffer);
    return IlTrue;
}

```

Initializing Panels

When the `initializePanels` method is called, the panel properties are loaded and the predefined panels are created, but the panel properties are not yet applied to the panels. Create your own panels in this method as follows:

```

IlBoolean
MyStudioExtension::initializePanels()
{
    IlvStudio* editor = getEditor();
    // Create MyGadgetPalette.
    MyGadgetPalette* pal = new MyGadgetPalette(editor);
    pal->connect();
    // Create MyPanel.
    MyPanelHandler* pan = new MyPanelHandler(editor, "MyPanel");
    pan->connect();
    return IlTrue;
}

```

Registering Inspectors

To register an inspector panel, you have to map an object that is able to create the inspector panel to the class name of the inspected object. The inspector panel builder must derive from the class `IlvStInspectorPanelBuilder`. To declare an inspector panel builder class, you must use the macro `IlvStDefineInspectorPanelBuilder`, as follows:

```

IlvStDefineInspectorPanelBuilder(MyClassInspector, \
                                MyClassInspectorBuilder)

```

The mapping used to register an inspector panel is done inside the `initializeInspectors` method, which is called after the predefined inspectors are initialized.

Here is how you add an inspector panel builder:

```

IlBoolean
MyStudioExtension::initializeInspectors()
{
    IlvStudio* editor = getEditor();
    editor->inspector().registerBuilder("MyClass",
                                     new MyClassInspectorBuilder);

    return IlTrue;
}

```

Initializing Editing Modes

The `initializeModes` method is called after the predefined editing modes are initialized. If you provide an editing mode, you can initialize it here.

Last Initialization Step

The `postInitialize` method is the last method to be called.

Integrating your Own Graphic Objects

This section explains how to integrate your own graphic object subclasses into IBM® ILOG® Views Studio.

Follow these steps to integrate a new class:

1. Define the command that will be used to add an instance of the class to the current buffer.
2. Declare the required `#include` statement to be generated in the panel class that uses your objects.
3. Put an instance of your class in the existing Palettes panel or provide your own palette.
4. Provide an Inspector panel to edit the properties of your objects.

Defining a New Command to Add an Object

To add an instance of a user-defined class to IBM® ILOG® Views Studio, you have to write a new command. The class `IlvStClickAddObject` defines a command that can be used to add an object at the position indicated by a mouse click. To create an instance of a user-defined graphic class, you will have to redefine its virtual member function `makeObject` in a derived class, as shown in the example below:

```
#include <ivstudio/edit.h>

class MyAddClass: public IlvStClickAddObject {
protected:
    virtual IlvStError* makeObject(IlvGraphic*& obj, IlvStudio* ed, IlvAny) {
        MyClass* mc = new MyClass(ed->getDisplay(), IlvRect(0, 0, 40, 40));
        obj = mc;
        return 0;
    }
};
```

The following command constructor creates and returns a new instance of the user-defined class:

```
static IlvStCommand*
MkMyAddClass(IlvStudio*)
{
    return new MyAddClass;
```

```
}

```

You can then create a subclass of `IlvStExtension` and define the `initializeCommandDescriptors` method to register the command constructor, as follows:

```
IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    getEditor()->registerCommand("AddMyClass", MkMyAddClass);
    return IlTrue;
}

```

In an option file, you can write your command declaration like this:

```
studio {
    // ...
    command AddMyClass {
        label "MyClass";
        prompt "Add an object of my class";
        category add;
    }
    // ...
}

```

The name given to the command is the same as the one registered with the editor. In this example, the command is displayed in the add category of the Commands panel.

If required, you can declare your option file using the `ILVSTOPTIONFILE` environment variable.

Adding the Include File and Library File of a New Class to the Generated Code

The C++ code that defines a panel containing an instance of a new user-defined class must contain the `#include` statement corresponding to the new class. To add this instruction, insert the following code in an initialization method of your extension class.

For example:

```
#include <ivstudio/appcode.h>

IlBoolean
MyStudioExtension::initializeBuffers()
{
    // If the IlvRegisterClass is not already done.
    This macro must be called only once.
    IlvRegisterClass (MyClass, TheSuperClass);
    IlvRegisterClassCodeInformation (MyClass,"<myclass.h>","mylib");
    // ...
    return IlTrue;
}

```

Customizing the Palettes Panel

You can add your own palettes to the Palettes panel using the `.opt` file. To add a new palette, define the node that corresponds to the new palette in the tree gadget, and provide the data file containing the objects in that palette. In the option file, you can specify the class of the container that is used to read and display data files. You can also remove a predefined palette and specify the palette that is selected by default.

The Palettes panel is split into two areas. The area at the top of the panel displays a tree gadget, while the bottom area displays a scrolled view. The tree gadget represents the hierarchy of available palettes from which you can choose. A container in the scrolled view displays the contents of the selected palette. Each node in the tree corresponds to a palette descriptor, which is defined by a name, a data file, a label, and the location of the node in the tree. The palette descriptor has its own container.

The container of a palette is created when the palette is selected for the first time. If the container class is specified, IBM® ILOG® Views Studio uses the corresponding container information (`IlvStContainerInfo`) to create an instance of the specified class. By default, it uses an `IlvGadgetContainer` object. If the palette has a specified data file, the created container reads that data file. All the palette containers are hidden, except the one that is attached to the selected palette.

Customize Options

You can use an option file to add new palettes to the Palettes panel, remove predefined palettes, or designate the default palette:

To describe a new palette, use the `dragDropPalette` option as follows:

```
dragDropPalette "<palette name>" {
    <option-1 <value-1>;
    ...
    <option-n <value-n>;
}
```

To remove a predefined palette from the tree gadget in the Palettes panel, use the `removeDragDropPalette` option like this:

```
removeDragDropPalette "<palette name>"
```

To specify the palette that is selected by default, use the `defaultDragDropPalette` option:

```
defaultDragDropPalette "<palette name>"
```

Broadcast Messages

Following are the messages that are broadcast when a palette container is initialized or when a palette is selected:

- ◆ `PaletteContainerInitialized` The argument is the descriptor of the selected palette. When this message is broadcast, the container is created, its data file is read and the objects in the palette are initialized.
- ◆ `PaletteSelected` The argument is the descriptor of the selected palette.

Example

The following example shows how to use options related to the IBM ILOG Views Studio palettes:

```
// mystudio.opt
studio {
  dragDropPalette "MyRootPalette" {
    dataFileName "myfile1.ilv";
    path -before "Gadgets";
  }
  dragDropPalette "MyPalette" {
    label "My Palette";
    bitmap "myicon.gif";
    dataFileName "myfile2.ilv";
    path "Gadgets" "Miscellaneous";
  }
  removeDragDropPalette "ViewRectangles";
  defaultDragDropPalette "MyRootPalette";
}
```

Defining and Integrating an Inspector Panel

To inspect the properties of a user-defined graphic object, do the following:

1. Define a new inspector panel class for this object class.
2. Integrate this inspector class into IBM® ILOG® Views Studio.

Defining an Inspector Panel Class

The new inspector panel class must derive from the `IlvStInspectorPanel` class, which is declared in the file `$(ILVHOME)/studio/ivstudio/inspectors/insppnl.h`. You could also derive this class from `IlvStIGraphicInspectorPanel` to automatically inherit from the inspection features of properties that are common to `IlvGraphic` objects. These properties will be displayed in two notebook pages: `General` and `Callbacks`. This class can be found in the file `$(ILVHOME)/studio/ivstudio/inspectors/gadpnl.h`. It defines an inspector panel that edits a subclass of an `IlvGraphic` class.

To derive this class, define the following functions:

- ◆ The constructor calls the parent class constructor providing a display, the title of the panel, a data file name, and the update mode. The last two parameters are optional.
- ◆ `initializeEditors` is called to register the accessors and editors in the inspector panel. Do not forget to declare the notebook pages that should appear in the inspector panel at the beginning of the method.

- ◆ `initWith` is called whenever the inspector panel is initialized with a new object. It initializes the panel according to the given object. For example, if you edit the x position of the object in a line editor, you set the label of the line editor to the position of the x object using this function. By default, this method initializes accessors and editors and should not be overridden.
- ◆ `applyChange` is called when the user clicks the Apply button to apply changes to the inspected object. By default, this method delegates his role to accessors and editors and should not be overridden.

The following is an example of these function definitions:

```
class MyClassInspector
: public IlvStIGraphicInspectorPanel
{
public:
    // Constructor
    MyClassInspector(IlvDisplay* display,
                    const char* title,
                    const char* filename = 0,
                    IlvSystemView transientFor = 0,
                    IlvStIAccessor::UpdateMode mode
                    = IlvStIAccessor::OnApply):
        IlvStIGraphicInspectorPanel(display, title, filename,
                                    transientFor, mode) {}
    virtual void initializeEditors();
};

IlvStDefineInspectorPanelBuilder(MyClassInspector, \
                                MyClassInspectorPanel);

MyClassInspector::MyClassInspector(IlvDisplay* display,
                                   const char* title,
                                   const char* filename,
                                   IlvSystemView transientFor,
                                   IlvStIAccessor::UpdateMode mode)
:IlvInspectorPanel(display, title, filename, transientFor, mode)
{}

void
MyClassInspector::initializeEditors()
{
    IlvStIGraphicInspectorPanel::initializeEditors();
    // Add notebook pages.
    addPage("&Specific", "../data/myclinsp.ilv");
    // Add editors.
    link("xfield", IlvGraphic::_xValue);
    link("yfield", IlvGraphic::_yValue);
}

```

Integrating the Inspector Panel into IBM ILOG Views Studio

To integrate the new inspector into the editor, modify your extension class to define the `initializeInspectors` as follows:

```
IlBoolean
```

```

MyStudioExtension::initializeInspectors()
{
    IlvStudio* editor = getEditor();
    editor->inspector().registerBuilder("MyClass",
                                     new MyClassInspectorPanel);
    return IlTrue;
}

```

Extending IBM ILOG Views Studio: An Example

The example in this section shows how to create an editor that associates predefined callbacks with any graphic object. `loadilv` is a predefined callback that takes the name of the file to be loaded as a parameter. This name is stored in graphic objects as a property.

The first task is to derive a class from `IlvGadgetManager` to redefine `read` and `write` for storing and restoring the new property. This part is not described here. We assume that you have a class `MyManager` that saves objects with the descriptor `MyGadgetManagerOutput` and a class `MyContainer` restoring objects saved by `MyManager`.

Follow these steps to extend the editor:

1. *Defining a New Buffer* associated with the new manager and container.
2. *Defining a New Command* to create a buffer of the new type.
3. *Defining a New Panel* to associate the file name with objects.

Defining a New Buffer Class

Define a subclass `MyGadgetBuffer` from `IlvStGadgetBuffer`. Below is a header example:

```

class MyGadgetBuffer
: public IlvStGadgetBuffer {
public:
    MyGadgetBuffer(IlvStudio*, const char* name, IlvManager* = 0);
    virtual const char* getType () const;
    virtual const char* getTypeLabel() const;
    virtual void setUpPanelClass(IlvStPanelClass*) const;
};

```

You provide:

- ◆ The constructor, which calls the `IlvStGadgetBuffer` constructor. If the manager parameter is not yet created, it creates a `MyManager` instance.
- ◆ The virtual member function `getType`, which returns the class name `MyGadgetBuffer`.

- ◆ The virtual member function `getTypeLabel`, which returns the class name label. This label is used by IBM ILOG Views Studio to display the type of the buffer in the Main window. It may be different from the label returned by the `getType` member function that is used as the identifier of the buffer type.
- ◆ The virtual member function `setUpPanelClass`, which is called when an IBM ILOG Views Studio panel class is made for your buffer.

The `MyGadgetBuffer` class can be defined like this:

```
#include <ivstudio/studio.h>
#include <ivstudio/stdesc.h>

#include <mybuf.h>
#include <myman.h>
#include <mycont.h>

MyGadgetBuffer::MyGadgetBuffer(IlvStudio* editor,
                               const char* name,
                               IlvManager* mgr)
: IlvStGadgetBuffer(editor,
                    name,
                    mgr ? mgr : new MyManager(editor->getDisplay()))
{
}

const char*
MyGadgetBuffer::getType () const
{
    return "MyGadgetBuffer";
}

const char*
MyGadgetBuffer::getTypeLabel () const
{
    return "Mine";
}

void
MyGadgetBuffer::setUpPanelClass(IlvStPanelClass* pclass) const
{
    IlvStGadgetBuffer::setUpPanelClass(pclass);
    pclass->setBaseClass("MyContainer");
}
```

Once you have the new class, you have to integrate it into the editor; that is, tell the editor that the file saved with the new descriptor needs to be loaded in the new buffer type.

To do so, add a call to `registerType` in your `initializeBuffers` method of your extension class. The following is an example:

```
#include <mybuf.h>

static IlvStBuffer*
MakeMyBuffer(IlvStudio* editor, const char* name, const char*)
{
```

```

        return new MyGadgetBuffer(editor, name);
    }

    IlvBoolean
    MyStudioExtension::initializeBuffers()
    {
        IlvStudio* editor = getEditor();
        // ...
        editor->buffers().registerType("MyGadgetManagerOutput",
                                     MakeMyBuffer);
        // ...
        return IlvTrue;
    }

```

Defining a New Command

The editor now recognizes what `MyManager` has generated. But a new buffer instance must be created.

To do so, provide a new command to create an instance of `MyBuffer`. Make a subclass of `IlvStCommand`, redefining the virtual member function `doIt`. The following is an example:

```

const char* NameNewBuffer = "MyNewBuffer";

class MyNewBuffer: public IlvStCommand {
public:
    virtual IlvStError* doIt(IlvStudio*, IlvAny);
};

IlvStError*
MyNewBuffer::doIt(IlvStudio* editor, IlvAny arg)
{
    if (arg) {
        editor->buffers().setCurrent((IlvStBuffer*)arg);
        return 0;
    }
    const char* name = editor->options().getDefaultBufferName();
    IlvStBuffer* buffer = new MyGadgetBuffer(editor, name);
    if (editor->buffers().get(name))
        buffer->newName(name); // uniq name
    return editor->execute(IlvNmNewBuffer, 0, 0, buffer);
}

```

Now the command must be integrated into the editor. To do so:

1. Add the registration of the new command to your initialize function, providing a function to build it.
2. Describe the new command in a new command declaration file named `mystudio.cmd`. You have to specify this command declaration file in your option file using the `commandFile` option.

The following is an example of the `initialize` function:

```

static IlvStCommand*
MkMyNewBuffer(IlvStudio*)

```

```

{
    return new MyNewBuffer;
}

IlBoolean
MyStudioExtension::initializeCommandDescriptors()
{
    IlvStudio* editor = getEditor();
    // ...
    editor->registerCommand("MyNewBuffer", MkMyNewBuffer);
    // ...
    return IlTrue;
}

```

The following is a command declaration example:

```

command MyNewBuffer {
    label "MyBuffer";
    prompt "Open my buffer";
    category buffer;
}

```

Defining a New Panel

Now you have to create a new panel to get the new property. Below are the steps to follow:

1. Create a panel using IBM® ILOG® Views Studio with a line editor named `file name` and two buttons with the callbacks `Apply` and `Cancel`.
2. Describe a subclass of `IlvStDialog` to provide the constructor with the member functions `Apply` and `Cancel`.
3. Integrate the new panel into the editor.

Here is a header example:

```

#include <ivstudio/panel.h>
class MyPanelHandler
: public IlvStDialog {
public:
    MyPanelHandler(IlvStudio* ed, const char* name,
                  IlvDialog* dlg = 0);
    virtual void apply();
    virtual void reset();
};

```

You provide:

- ◆ The constructor, which calls `IlvStDialog` constructor giving the data file you created. It initializes the panel and subscribes to the message `ObjectSelected` by the member function `resetOnMessage`. The callback passed to the subscription calls the `reset` member function for the panel.
- ◆ The virtual member function `apply`, which reads the file name object contents and associates it with the object property.

- ◆ The virtual member function `reset`, which initializes the file name object contents for the property of the currently selected object.

The following is a coding example:

```
#include <ivstudio/studio.h>
#include <mypan.h>
#include <myutil.h>
#define DATAFILE "../data/mypanel.ilv"

MyPanelHandler::MyPanelHandler(IlvStudio* ed, const char* name,
                               IlvDialog* dlg)
: IlvStDialog(ed, name, DATAFILE, IlvRect(0, 0, 254, 71))
{
    IlvTextField* tf =
        (IlvTextField*)getDialog()->getObject("filename");
    tf->setLabel("", IlTrue);
    resetOnMessage("ObjectSelected");
}

void
MyPanelHandler::apply()
{
    IlvGraphic* obj = getEditor()->getSelection();
    if (obj) {
        const char* name =
            ((IlvTextField*)getDialog()->getObject("filename"))->getLabel();
        if (name && name[0]) {
            MySetParameter(obj, IlvGetSymbol(name));
            obj->setCallbackName(IlvGetSymbol("loadilv"));
        }
    }
}

void
MyPanelHandler::reset()
{
    IlvTextField* tf =
        (IlvTextField*)getDialog()->getObject("filename");
    IlvGraphic* obj = getEditor()->getSelection();
    IlvSymbol* fi = 0;
    if (obj)
        fi = MyGetParameter(obj);
    tf->setLabel(fi ? fi->name() : "", IlTrue);
}
```

Once the panel class is created, it must be integrated into the editor. To do so:

1. Add the building of the panel to the editor initialization function.
2. Provide its description in the file `mystudio.pnl`.

The following is a coding example with a command to display the panel.

```
static IlvStCommand*
MkMyShowPanel(IlvStudio* editor)
{
    return new IlvStShowPanel(editor->getPanel("MyPanel"));
}
```



```

IlBoolean
MyStudioExtension::initializePanels()
{
    // ...
    // Create MyPanel.
    MyPanelHandler* pan = new MyPanelHandler(getEditor(), "MyPanel");
    pan->connect();
    // ...
    return IlTrue;
}

```

Providing Container Information

To integrate your container class, you have to define a subclass of `IlvStContainerInfo` and add it to the IBM® ILOG® Views Studio information set, as shown below:

```

class MyContainerInfo
: public IlvStContainerInfo {
public:
    MyContainerInfo() : IlvStContainerInfo("MyContainer") {}
    IlvContainer* createContainer(IlvAbstractView* parent,
                                const IlvRect&   bbox,
                                IlBoolean        useacc,
                                IlBoolean        visible) {
        return new MyContainer(parent, bbox, useacc, visible);
    }
    IlvContainer* createContainer(IlvDisplay*   display,
                                const char*     name,
                                const char*     title,
                                const IlvRect&  bbox,
                                IlUInt         properties,
                                IlBoolean       useacc,
                                IlBoolean       visible,
                                IlvSystemView   transientFor) {
        return new MyContainer(display,
                                name,
                                title,
                                bbox,
                                properties,
                                useacc,
                                visible,
                                transientFor);
    }
    const char* getFileCreatorClass() const {
        return "MyGadgetManagerOutput";
    }
};

// ...
editor->addContainerInfo(new MyContainerInfo());

```

Registering Callbacks

IBM® ILOG® Views Studio lets you use your own callbacks when you test panels or applications by calling the `IlvStudio::registerCallback` function:

```
static void ILVCALLBACK
MyCallback(IlvGraphic* obj, IlvAny)
{
    IlvPrint("MyCallback is called");
}

....
IlvStudio* editor = ...
....
editor->registerCallback("MyCallback", MyCallback);
editor->registerCallback("myCallback", MyCallback);
....
```


Using Inspector Classes

This chapter introduces you to the use of the IBM® ILOG® Views Studio inspector classes. You can find information on the following topics:

- ◆ *What Is an Inspector?*
- ◆ *Components of an Inspector Panel*
- ◆ *Defining a New Inspector Panel*

For a more detailed description of the classes that are referred to in this chapter, see the IBM ILOG Views *Studio Reference Manual*.

Note: To use the inspector classes of IBM ILOG Views Studio, you need the IBM ILOG Views Gadgets and IBM ILOG Views Manager packages.

What Is an Inspector?

In IBM® ILOG® Views Studio, an inspector is an instance of the class `IlvStInspector`. IBM ILOG Views Studio contains one instance of this class, which is used to inspect selected graphic objects in the active buffer. The role of an inspector is to display the inspector panel that corresponds to the last selected graphic object.

To display the appropriate inspector panel, the inspector maintains a table that maps graphic object classes to inspector panel classes. If a graphic object class has no associated inspector panel, the inspector attaches it to the inspector panel of the first superclass in the inheritance path that has an associated inspector panel. Let us suppose that the object to be inspected is of the type `IlvMyTextField`, a class derived from `IlvTextField`. If no inspector panel has been defined for this class, the IBM ILOG Views Studio inspector displays the `IlvTextField` inspector panel.

An inspector panel is made up of several components, which are described in the following sections.

Components of an Inspector Panel

Inspecting an object boils down to examining its properties. In general, to inspect a property, an inspector panel uses the following pair of components: an *accessor* and an *editor*.

The accessor interfaces with the inspected property while the editor interfaces with a gadget that represents it graphically in the inspector panel (an `IlvTextField`, for example). In the context where an accessor is paired with an editor, the accessor is responsible for fetching the property value and displaying it via the editor. The editor for its part notifies the accessor whenever its content changes. In other words, inspecting a property means initializing the accessor when the inspector is initialized, and requesting the accessor to apply modifications made to the editor's content. In this context, only a list of accessors is required to inspect an object.

Certain editors, however, do not need to be linked with accessors to work. For example, a combo box used to show or hide a set of gadgets does not need to access data to be initialized. Similarly, changing the selected item in the combo box does not affect the data. Because these stand-alone editors are not initialized by accessors, they must be initialized explicitly.

To handle both the pairs accessors/editors and stand-alone editors, an inspector panel makes use of a main editor defined by the class `IlvStIMainEditor`. Actually, inspection operations, including managing the Apply button present in each inspector panel, which are carried out by the inspector panel, are processed by the main editor.

The following figures illustrate respectively:

- ◆ The various components of an inspector panel and how they relate to one another.
- ◆ The initialization steps of an inspector panel.
- ◆ What happens when a property is modified in a inspector panel.
- ◆ The steps involved in applying changes to properties made via an inspector panel.

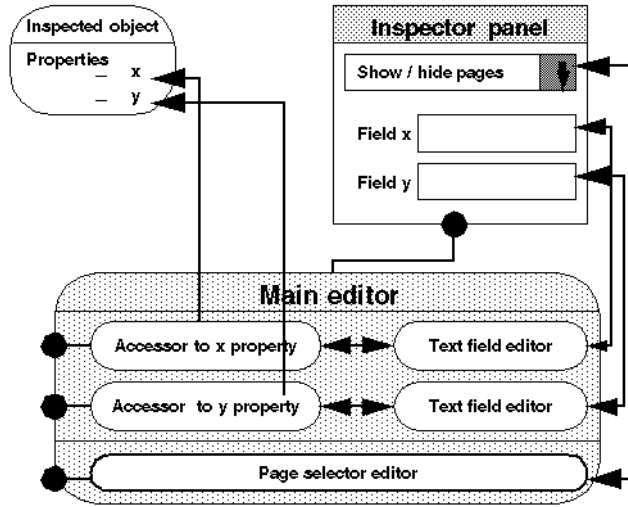
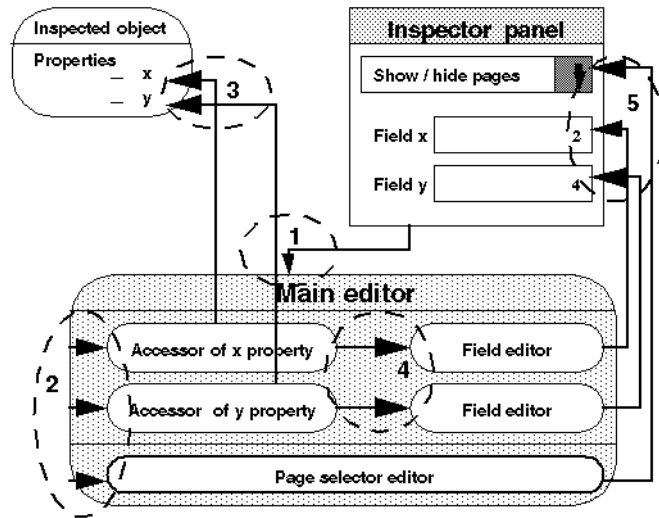
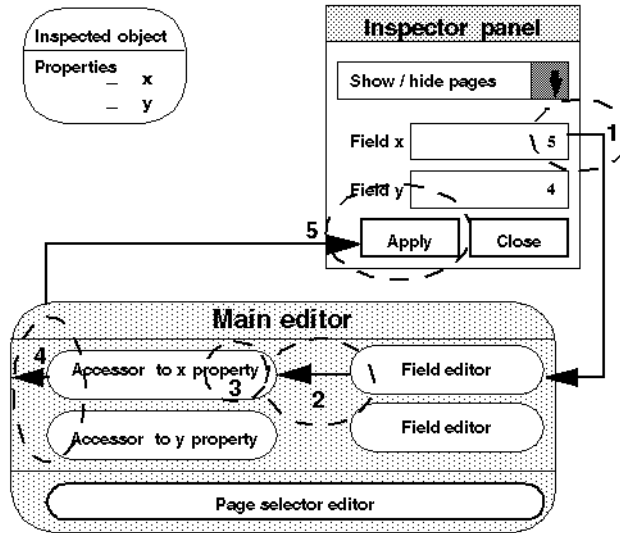


Figure 7.1 Components of an Inspector Panel



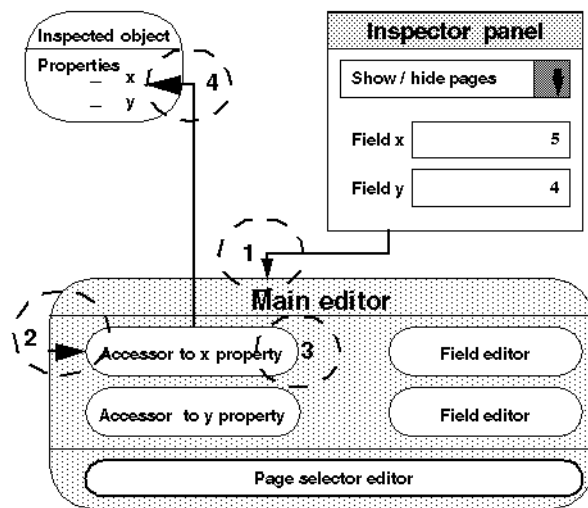
1. The main editor is initialized.
2. The main editor initializes its accessors and its stand-alone editors.
3. Accessors get property values.
4. Accessors initialize editors according to returned values.
5. Editors initialize gadgets.

Figure 7.2 Initialization Steps of an Inspector Panel



1. The user changes the value of the x text field.
2. The text field editor is notified about the user's modification and sets its associated accessor to the new value.
3. The accessor to the x property stores the new value and sets itself as modified.
4. The main editor is notified about the modification of one of its accessors and sets itself as modified.
5. The main editor changes the state of the Apply button when it is modified.

Figure 7.3 What Happens When Modifications are Made in the Inspector Panel



1. The user clicks the Apply button.
2. The apply() function of the main editor is called.
3. The main editor asks that apply() be called only for modified accessors.
4. The modified accessor applies the new value to the inspected object.
5. Once modifications have been applied, the accessor is no longer in the modify state.

Figure 7.4 Applying Modifications Made in an Inspector Panel

Accessors

An inspector panel handles accessors of the class `IlvStIAccessor`, which is the base class of all the accessor classes. It performs two actions on accessors by calling the methods `initialize` and `apply`. Calling the first method initializes the calling accessors, while `apply` brings into effect the modifications made to the inspected object.

Property Accessors

Most of the time, accessors are used to inspect object properties. In fact, if you take a look at the accessor class hierarchy illustrated below, you'll see that `IlvStIPropertyAccessor`, a subclass of `IlvStIAccessor`, is the base class for all types of accessors in the library.

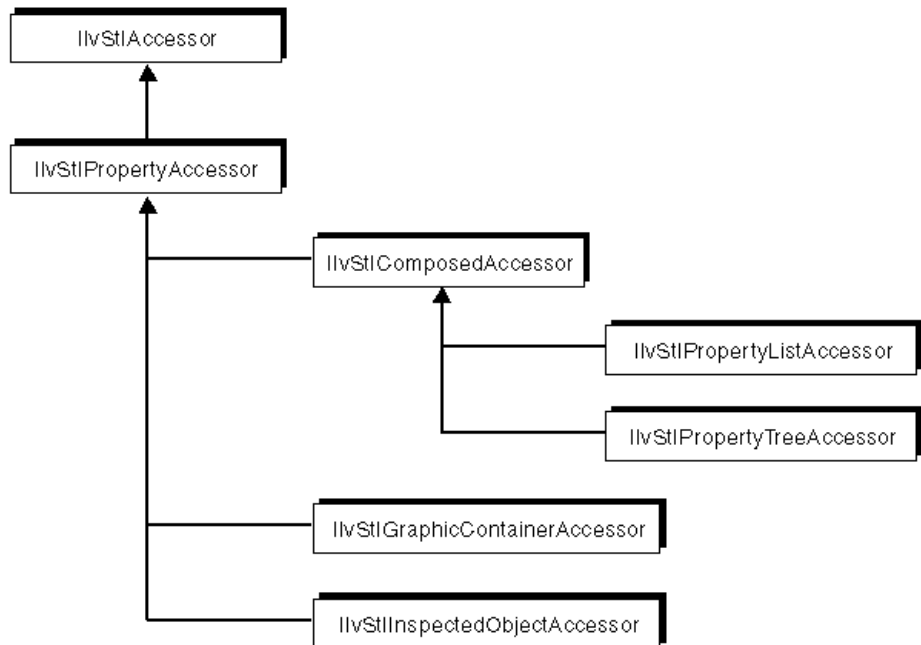


Figure 7.5 Accessor Hierarchy

Property accessors manipulate properties via the class `IlvStIProperty` in which they are encapsulated. For example, to manipulate a property of the type `IlvValue`, an accessor uses an object of the type `IlvStIValueProperty` deriving from the class `IlvStIProperty`, in which the `IlvValue` object is encapsulated.

Accessors inspect properties using two different modes:

- ◆ An update mode, which specifies whether the property accessor should apply modifications immediately or when the user clicks the Apply button.
- ◆ A building mode, which specifies whether a property should be created, if not found, and/or copied.

Since the `initialize` and `apply` methods of the property accessor utilize these parameters, you must not redefine them when subclassing `IlvStIPropertyAccessor`. Instead redefine the methods `getOriginalValue` and `applyValue`, which are invoked by `initialize` and `apply`, respectively.

The following example shows how to subclass `IlvStIPropertyAccessor` to access the label of a gadget item:

```
class IlvLabelAccessor
: public IlvStIPropertyAccessor
```

```

{
public:
    IlvLabelAccessor(IlvGadgetItem* gadgetItem,
                    const char* name = 0,
                    UpdateMode updateMode = NoUpdate,
                    BuildMode buildMode = None):
        IlvStIPropertyAccessor(name, updateMode, buildMode),
        _gadgetItem(gadgetItem)
    {}
protected:
    IlvGadgetItem* _gadgetItem;
    IlvGadgetItem* getGadgetItem()const;
    virtual IlvStProperty* getOriginalValue() const;
    virtual void applyValue(IlvStProperty* property);
};

IlvGadgetItem*
IlvLabelAccessor::getGadgetItem() const
{
    return _gadgetItem;
}

IlvStIProperty*
IlvLabelAccessor::getOriginalValue()
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    return new IlvStIValueProperty(gadgetItem->getLabel(), "label");
}

void
IlvLabelAccessor::applyValue(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    IlvValue value;
    property->getValue(value);
    const char* label = (const char*)value;
    gadgetItem->setLabel(label);
}

```

Dependent Accessors

Certain inspected properties directly depend on other inspected properties. For example, the user should not be able to inspect the intermediate state of a toggle button if the intermediate mode was not set for it. In other words, the accessor to the “intermediate state” property should always be aware of the value set for the accessor to the “intermediate mode” property, and its associated editor should appear gray or not depending on that value. This means that if the accessor to the “intermediate mode” property is initialized or is modified, the accessor to the “intermediate state” must be reinitialized accordingly. For this initialization precedence order to be achieved, the accessor to the “intermediate state” property should be made dependent on the accessor to the “intermediate mode” property using the method `IlvStIAccessor::addDependentAccessor`.

This dependency mechanism is also used by combined accessors, which are described in the next section.

Combined Accessors

A combined accessor is an instance of the class `IlvStICombinedAccessor`, a subclass of `IlvStIPropertyAccessor`, which is used to inspect the property of an object that is returned by another accessor. For example, let us consider a name accessor used to inspect the name of a gadget item. By combining this accessor with a gadget item accessor, you can use it to inspect both an element selected in a gadget item list or a gadget item in a message label. Combined accessors are usually implemented as dependent accessors since they must be reinitialized whenever the property they access through another accessor is itself reinitialized. In our example, changing the current selection in a gadget item list would cause the name accessor to be reinitialized.

The example given in the section *Accessors* on page 169 has been rewritten below to illustrate combined accessors. It shows how to subclass `IlvStICombinedAccessor` to access the label of a gadget item:

```
class IlvLabelAccessor
: public IlvStICombinedAccessor
{
public:
protected:
    IlvGadgetItem* getGadgetItem() const;
    virtual IlvStProperty* getOriginalValue();
    virtual void applyValue(IlvStProperty* property);
};

IlvGadgetItem*
IlvLabelAccessor::getGadgetItem()
{
    if (!getObjectAccessor())
        return 0;
    IlvStIProperty* property = getObjectAccessor()->get();
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

// The implementation of the getOriginalValue and applyValue methods
// are the same as in previous the sample.
...
```

List Accessors

A list accessor is an instance of the class `IlvStIPropertyListAccessor`, which derives from `IlvStICombinedAccessor`. A list accessor is used to inspect a list of properties. It allows you to add, remove, or modify a property in a list. This type of accessor works in conjunction with instances of the class `IlvStIPropertyListEditor`. Editors of this kind handle gadgets that are used to edit lists, that is, list gadgets, and the following four buttons: Add After, Add Before, Remove, and Clean.

The following code sample shows how to access a list of gadget items that are contained in a gadget item holder:

```
class IlvStIGadgetItemListAccessor
: public IlvStIPropertyListAccessor {
```

```

public:
    // -----
    // Constructor / destructor
    IlvStIGadgetItemListAccessor(IlvStIPropertyAccessor* accessor = 0,
                                IlvStIAccessor::UpdateMode updateMode=
                                    IlvStIAccessor::Inherited,
                                const char* name = 0);
    ~IlvStIListGadgetItemAccessor();

    IlvListGadgetItemHolder* getListGadgetItemHolder() const;

protected:
    IlvGadgetItem* getGadgetItem(const IlvStIProperty*) const;
    virtual IlvStIProperty** getInitialProperties(ILUInt& count);
    virtual IlvStIProperty* createDefaultProperty() const;
    virtual IlvGadgetItem* createGadgetItem(
        const IlvStIProperty* prop) const;

    virtual void addProperty(IlvStIProperty* property, ILUInt index);
    virtual void replaceProperty(IlvStIProperty* origProperty,
                                IlvStIProperty* newProperty,
                                ILUInt index);
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property, ILUInt index);
    virtual void moveProperty(IlvStIProperty* property,
                              ILUInt previousIndex,
                              ILUInt newIndex);
};

IlvStIGadgetItemListAccessor::
    IlvStIGadgetItemListAccessor(IlvStIPropertyAccessor* accessor,
                                IlvStIAccessor::UpdateMode updateMode,
                                const char* name):
    IlvStICombinedAccessor(accessor, update, name)
{
}

IlvStIGadgetItemListAccessor::~IlvStIGadgetItemListAccessor()
{
}

IlvListGadgetItemHolder*
IlvStIGadgetItemListAccessor::getListGadgetItemHolder() const
{
    if (!getObjectAccessor())
        return 0;
    IlvStIProperty* property = getObjectAccessor()->get();
    return (property? (IlvListGadgetItemHolder*)property->get() : 0);
}

IlvStIProperty**
IlvStIListGadgetItemAccessor::getInitialProperties(ILUInt& count)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return 0;
    count = (ILUInt)listHolder->getCardinal();
    if (!count)
        return 0;
}

```

```

    IlvStIProperty** properties = new IlvStIProperty*[count];
    for(IlUInt i = 0; i < count; i++)
        properties[i] = new IlvStIValueProperty(
            (IlvAny)listHolder->getItem((IlvUShort)i));
    return properties;
}

IlvGadgetItem*
IlvStIListGadgetItemAccessor::getGadgetItem(
    const IlvStIProperty* property) const
{
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

IlvStIProperty*
IlvStIListGadgetItemAccessor::createDefaultProperty() const
{
    return new IlvStIValueProperty(
        (IlvAny)new IlvGadgetItem("&Item", (IlvBitmap*)0));
}

IlvGadgetItem*
IlvStIListGadgetItemAccessor::createGadgetItem(
    const IlvStIProperty* prop) const
{
    const IlvStIGadgetItemValue* value =
        ILVI_CONSTDOWNCAST(IlvStIGadgetItemValue, prop);
    if (!value)
        return 0;
    IlvGadgetItem* newGadgetItem =
        (value->getGadgetItem()? value->getGadgetItem()->copy() : 0);
    if (!newGadgetItem)
        return 0;
    newGadgetItem->setSensitive(IlTrue);
    newGadgetItem->showLabel(IlTrue);
    newGadgetItem->showPicture(IlTrue);
    newGadgetItem->setEditable(IlFalse);
    return newGadgetItem;
}

void
IlvStIListGadgetItemAccessor::addProperty(IlvStIProperty* property,
    IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (listHolder) {
        listHolder->insertItem(getGadgetItem(property), (IlvShort)index);
    }
}

void
IlvStIListGadgetItemAccessor::replaceProperty(IlvStIProperty* origProperty,
    IlvStIProperty* newProperty,
    IlUInt position)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
}

```

```

        listHolder->removeItem((IlvUShort)position);
        listHolder->insertItem(getGadgetItem(newProperty), (IlvUShort)position);
    }

void
IlvStIListGadgetItemAccessor::deleteNewProperty(IlvStIProperty* property)
{
    delete getGadgetItem(property);
}

void
IlvStIListGadgetItemAccessor::deleteProperty(IlvStIProperty* property,
                                              IlUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvShort)(IlvUShort)index);
}

void
IlvStIListGadgetItemAccessor::moveProperty(IlvStIProperty* property,
                                           IlUInt previousIndex,
                                           IlUInt newIndex)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvUShort)previousIndex, IlFalse);
    listHolder->insertItem(getGadgetItem(property),
                          (IlvShort)(IlvUShort)(newIndex -
                                                  (newIndex > previousIndex? 1 : 0)));
}

```

Tree Accessors

A tree accessor is an instance of the class `IlvStIPropertyTreeAccessor`, which derives from `IlvStICombinedAccessor`. A tree accessor is used to inspect a tree of properties. It allows you to add, remove, or modify a property in a tree. This type of accessors works in conjunction with instances of the class `IlvStIPropertyTreeEditor`. Editors of this kind handle gadgets that are used to edit trees, that is, tree gadgets, and the following five buttons: Add After, Add Before, Add Child, Remove, and Clean.

The following code sample shows how to access a tree of gadget items that are contained in a tree gadget:

```

class IlvStIGadgetItemTreeAccessor
: public IlvStIPropertyTreeAccessor {
public:
    IlvStIGadgetItemTreeAccessor(IlvStIPropertyAccessor* accessor = 0,
                                 IlvStIAccessor::UpdateMode updateMode =
                                 IlvStIAccessor::Inherited,
                                 const char* name = 0,
                                 IlvStIAccessor::BuildMode buildMode =
                                 IlvStIAccessor::Copy);
    ~IlvStIGadgetItemTreeAccessor();
}

```

```

// -----
IlvTreeGadgetItemHolder* getTreeGadgetItemHolder() const;

protected:

    IlvTreeGadgetItem* getGadgetItem(const IlvStIProperty*) const;
    IlvTreeGadgetItem* getParentGadgetItem(const IlvStIProperty*) const;

// Applying.

    virtual IlUInt getChildPosition(const IlvStIProperty* parentProperty,
                                    const IlvStIProperty* property) const;
    virtual void addProperty(IlvStIProperty* property,
                             const IlvStIProperty* parent,
                             IlUInt childPosition);
    virtual void replaceProperty(IlvStIProperty* origProperty,
                                 IlvStIProperty* newProperty,
                                 const IlvStIProperty* parent,
                                 IlUInt childPosition);

// Array of properties.
    virtual IlvStIProperty** getInitialChildrenProperties(
                                    IlUInt& count,
                                    const IlvStIProperty* parent = 0) const;

// Insertion of properties.
    virtual IlvStIProperty* createProperty(const IlvStIProperty* parent,
                                           IlUInt childPosition,
                                           IlvAny param = 0) const;

// Destruction of properties.
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property);
};

IlvStIGadgetItemTreeAccessor::IlvStIGadgetItemTreeAccessor(
    IlvStIPropertyAccessor* accessor,
    IlvStIAccessor::UpdateMode updateMode,
    const char* name,
    IlvStIAccessor::BuildMode buildMode):
    IlvStIPropertyTreeAccessor(accessor,
                                updateMode,
                                buildMode,
                                (name? name : "GadgetItemTreeAccessor"))
{
}

IlvStIGadgetItemTreeAccessor::~IlvStIGadgetItemTreeAccessor()
{
}

IlvTreeGadgetItemHolder*
IlvStIGadgetItemTreeAccessor::getTreeGadgetItemHolder()const
{
    IlvStIProperty* property = (_accessor? _accessor->get() : 0);
    return (property? (IlvTreeGadget*)property->getPointer() : 0);
}

// -----
IlvTreeGadgetItem*

```



```

IlvStIGadgetItemTreeAccessor::getGadgetItem(
    const IlvStIProperty* property) const
{
    return (property? (IlvTreeGadgetItem*)property->getPointer() : 0);
}

IlvTreeGadgetItem*
IlvStIGadgetItemTreeAccessor::getParentGadgetItem(
    const IlvStIProperty* property) const
{
    if (!property) {
        // Returns root.
        IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
        if(!holder)
            return 0;
        return holder->getRoot();
    }
    return (property? (IlvTreeGadgetItem*)property->getPointer() : 0);
}

IlUInt
IlvStIGadgetItemTreeAccessor::getChildPosition(
    const IlvStIProperty* parentProperty,
    const IlvStIProperty* property) const
{
    // Get parentItem.
    IlvTreeGadgetItem* parentItem = getParentGadgetItem(parentProperty);
    if (!parentItem)
        return (IlUInt)-1;

    IlvTreeGadgetItem* findItem = getGadgetItem(property);
    IlUInt position = 0;
    for(IlvTreeGadgetItem* item = parentItem->getFirstChild();
        item;
        item = item->getNextSibling(), position++) {
        if (item == findItem)
            return position;
    }
    return (IlUInt)-1;
}

void
IlvStIGadgetItemTreeAccessor::addProperty(IlvStIProperty* property,
    const IlvStIProperty* parent,
    IlUInt index)
{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if (!holder)
        return;
    holder->addItem(getParentGadgetItem(parent),
        getGadgetItem(property), (IlvInt)index);
}

void
IlvStIGadgetItemTreeAccessor::replaceProperty(IlvStIProperty* origProperty,
    IlvStIProperty* newProperty,
    const IlvStIProperty* property,
    IlUInt index)

```

```

{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if (!holder)
        return;
    // Instead of removing the old gadget item and adding the new one, the
    // following line copies the attributes of the new created gadget item
    // to the old one.
    *(getGadgetItem(origProperty)) = *getGadgetItem(newProperty);
    // After this method is called, newProperty becomes the new
    // original property and should therefore be updated.
    // As we have copied attributes from the new created gadget item
    // to the initial one, the inspected gadget item
    // keeps being the one contained in origProperty.
    newProperty->setPointer(origProperty->getPointer());
}

// Array of properties.
IlvStIProperty**
IlvStIGadgetItemTreeAccessor::getInitialChildrenProperties(
    IUInt& count,
    const IlvStIProperty* parent) const
{
    IlvTreeGadgetItem* parentItem = getParentGadgetItem(parent);
    if (!parentItem)
        return 0;
    IlvArray properties;
    for(IlvTreeGadgetItem* item = parentItem->getFirstChild();
        item;
        item = item->getNextSibling()) {
        properties.add(new IlvStIValueProperty((IlvAny)item));
    }
    count = properties.getLength();
    if (!count)
        return 0;
    IlvStIProperty** props = new IlvStIProperty*[count];
    ::memcpy(props,
        properties.getArray(),
        (size_t)(sizeof(IlvStIProperty*) * (IlvInt)count));
    return props;
}

// Inserting properties.
IlvStIProperty*
IlvStIGadgetItemTreeAccessor::createProperty(const IlvStIProperty*,
    IUInt,
    IlvAny) const
{
    return new IlvStIValueProperty((IlvAny)new IlvTreeGadgetItem("&Item"));
}

// Destruction of properties.
void
IlvStIGadgetItemTreeAccessor::deleteNewProperty(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = (IlvGadgetItem*)property->getPointer();
    if (gadgetItem)
        delete gadgetItem;
}

```

```

void
IlvStIGadgetItemTreeAccessor::deleteProperty(IlvStIProperty* property)
{
    IlvTreeGadgetItemHolder* holder = getTreeGadgetItemHolder();
    if (!holder)
        return;
    holder->removeItem(getGadgetItem(property));
}

```

Preconditions and Validators

Accessors use internally the classes `IlvStIPrecondition` and `IlvStIValidator`, and their derived classes, to perform a number of verifications.

Preconditions

Preconditions are tests that accessors can run to determine whether they can access the inspected property. These tests are performed by calling the method `isAccessible` of the class `IlvStIPrecondition`. If the precondition test succeeds, access is allowed. Otherwise, it is denied and the associated editors are disabled.

The classes `IlvStIPreconditionValue` and `IlvStICallbackPrecondition` are two classes derived from `IlvStIPrecondition` that are sufficient to run precondition tests in most cases.

`IlvStIPreconditionValue` objects compare the value returned by an accessor with a given value.

Let us consider an accessor to the scientific mode property of a number field. As shown below, access is permitted only if the float mode is set for the number field.

```

//This code extract is part of the code of an inspector panel.
IlvStIEditor* editor = link("NumFieldFloat", IlvNumberField::_floatModeValue);
IlvStIPropertyAccessor* floatAccessor = editor->getAccessor();
floatAccessor->setPreviewValueAccessor(previewAccessor,
                                       IlvNumberField::_floatModeValue);

// Scientific value.
IlvStIEditor* editor = link("ScientificField",
                            IlvNumberField::_scientificModeValue);
editor->getAccessor()->setPrecondition(
    new IlvStIPreconditionValue(floatAccessor,
                                (IlvBoolean)IlvTrue,
                                (IlvBoolean)IlvFalse));

```

The `IlvStICallbackPrecondition` class is provided for cases where the code of the `isAccessible` function can be included in a callback. Using this class, you can avoid deriving the `IlvStIPrecondition` class.

The following code sample implements a precondition that is used to avoid changing the alignment of a gadget item label if it does not contain one or more “end-of-line” characters.

```

IlvBoolean

```

```

IlvStIsMultiLineText(IlvStIProperty* property,
                    IlvAny,
                    IlvStIProperty**,
                    IlvStIPropertyAccessor::PropertyStatus*)
{
    if (!property)
        return IlFalse;
    IlvValue value;
    const char* label = (const char*)property->getValue(value);
    if (!label)
        return IlFalse;
    while (*label)
        if (*label++ == '\n')
            return IlTrue;
    return IlFalse;
}

```

The callback precondition will be used as follows:

```

// Define an accessor to the label of the inspected gadget item.
IlvStIPropertyAccessor* labelAcc;
...
// Define the accessor to the alignment of the
// the inspected gadget item label.
IlvStIPropertyAccessor* labelAlignAcc;
...
labelAlignAcc->setPrecondition(
    new IlvStICallbackPrecondition(labelAcc,
                                   IlvStIsMultiLineText));

```

Validators

The IBM ILOG Views Studio Inspectors API includes a validator class, `IlvStIValidator`, that you can use to test whether the values entered by the user are correct. The test is performed with the `isValid` method of the class. This method tests the value passed as its parameter and returns an error of the type `IlvStIError` if the value is not valid. You can define whether the test should be carried out when the user's modifications are entered or only when he/she clicks on Apply button in the inspector panel.

`IlvStIValidator` has a derived class, `IlvStIRangeValidator`, that tests whether a value is between a minimum and a maximum value. In addition to a value range, this class takes a message string as a parameter. This message string specifies an error message that can contain one or more `%1`, `%2`, and `%3` substrings. These substrings are replaced by the minimum value, the maximum value, and the tested value, respectively.

The following example shows how to use a validator to check whether the month entered by the user is between 1 and 12.

```

// Define an accessor to the month property, called monthAccessor.
IlvStIPropertyAccessor* monthAccessor;
...
// Add the month validator to the month accessor.
IlvStIRangeValidator* monthValidator =
    new IlvStIRangeValidator((IlvInt)1, (IlvInt)12, "&MonthNotInRange");
monthAccessor->setValidator(monthValidator);

```

The message string "&MonthNotInRange" is translated as follows:

```
"You must specify a month between %1 and %2".
```

Editors

Editors are objects of the type `IlvStIEditor` used to edit inspected values.

Editors Associated with Accessors

In most cases, the value to be inspected is directly retrieved by its accessor and is then modified via the associated editor graphically represented by a gadget.

Here is a list of the gadget classes that can be associated with an editor:

- ◆ `IlvTextField`
- ◆ `IlvNumberField`
- ◆ `IlvToggle`
- ◆ `IlvStringList`
- ◆ `IlvOptionMenu`
- ◆ `IlvScrolledComboBox`
- ◆ `IlvSelector`
- ◆ `IlvSpinBox`

There is one class of editor for each one of the gadget classes enumerated above. These editor classes are encapsulated in the class `IlvStIDefaultEditorBuilder`, a subclass of `IlvStIEditor`, and are therefore transparent for the user. If you want to create an editor and associate it with a gadget, you have to build an instance of the class `IlvStIDefaultEditorBuilder` and provide it with the name of the gadget that you want to attach to it. When this instance is initialized, it creates an editor that corresponds to the type of the specified gadget. The created editor is managed as a child editor of the `IlvStIDefaultEditorBuilder` instance.

In the following example, we have an accessor, represented by the `floatAccessor` variable, that is used to inspect the float mode of a number field. Here is how you would create an editor and associate it with this accessor to handle a toggle named "floatToggle" in an inspector panel.

```
IlvStIEditor* editor =  
    new IlvStIDefaultEditorBuilder("floatToggle", floatAccessor);  
addEditor(editor);
```

Editors Not Associated with Accessors

In certain rare cases, inspected values can be so complex that they cannot be handled by an `IlvStIPropertyAccessor` object. It would be easier, for example, to fetch an array of

values directly from the inspected object and place it in a matrix rather than going through an `IlvStIPropertyAccessor` and trying to fit the value array in an `IlvStIProperty` object. In this case, you have to derive a class from `IlvStIEditor` and redefine the following virtual methods:

```

◆ virtual IlBoolean initialize() = 0;
◆ virtual IlBoolean apply() = 0;
◆ virtual IlBoolean connectHolder(IlvGraphicHolder* holder);
◆ virtual IlBoolean isModified() const;
◆ virtual void setModified(IlBoolean = IlTrue);

```

For more information about these methods, see the *IBM ILOG Views Studio Reference Manual*.

Instances of these derived editors are added to an inspector panel like any other editors by calling the method `IlvStIEditorSet::addEditor`.

Defining a New Inspector Panel

The following sections explain how to define a new inspector panel. Defining a new inspector panel involves two main steps that are detailed below. These are:

1. Create a new inspector class.
2. Incorporate the inspector class that was created to IBM® ILOG® Views Studio. This step is not covered in this chapter. For instructions on how to incorporate an inspector to IBM ILOG Views Studio, see *Registering Inspectors* on page 150.

The explanations in this section are based on an example, which we introduce in the next section.

Example

The example consists of creating the inspector panel for a combo box that displays a set of colors from which the user can choose. This inspector panel will be used to define the colors present in the combo box and configure the way these colors will be displayed.

Figure 7.6 shows the combo box and Figure 7.7 shows the associated inspector panel.

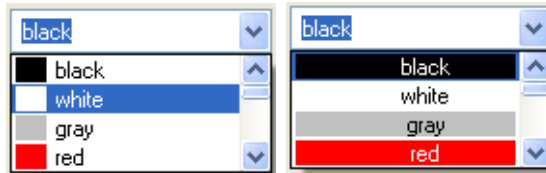


Figure 7.6 Color Combo Boxes with Small Color Rectangle (left) and Full Color Rectangle (right)

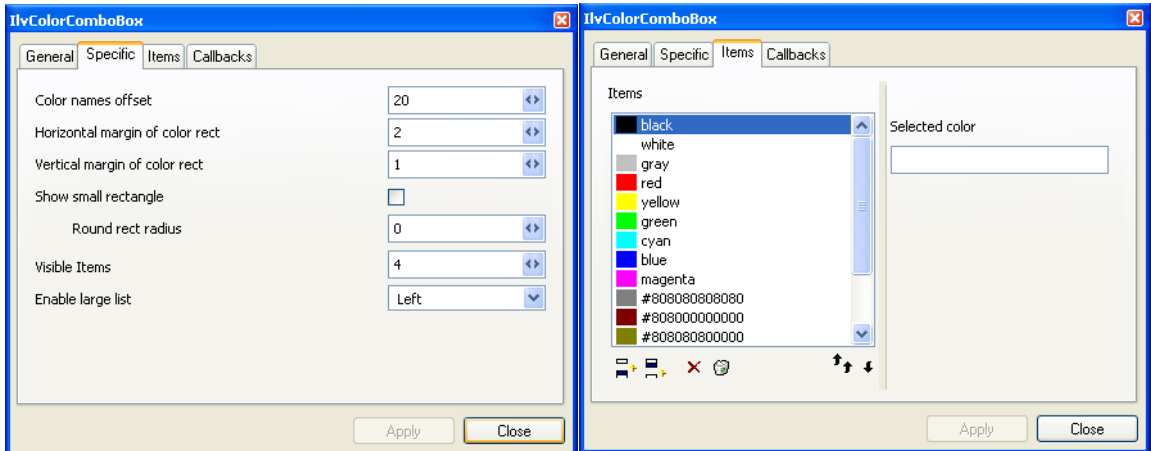


Figure 7.7 Pages of the Color Combo Box Inspector Panel

The complete code for this example can be found in the following directory:

`$(ILVHOME) / samples / studio / colorbox`

Creating the Color Combo Box Inspector Panel

Graphic object inspectors derive from the class `IlvStInspectorPanel`. Since in our example, the inspected graphic object is also a gadget, the inspector panel we are going to create derives from `IlvStIGadgetInspectorPanel`, a subclass of `IlvStInspectorPanel`.

```
class IlvColorComboBoxInspectorPanel
: public IlvStIGadgetInspectorPanel {
public:
    IlvColorComboBoxInspectorPanel(IlvManager* manager,
                                   IlvSystemView transientFor = 0,
                                   IlvStIAccessor::UpdateMode =
                                       IlvStIAccessor::OnApply);

    virtual void initializeEditors();
};
```

As we said in the section *Components of an Inspector Panel* on page 165, inspector panels are implemented with accessors and editors. These must be declared in the method `initializeEditors`. In our example, the definition of this method is divided in two parts, corresponding each to the implementation of the two notebook pages that make up the inspector panel (see Figure 7.7). These pages are described in the following two sections.

Note: *The General and Callback pages are created automatically.*

Implementing the Specific Page

The Specific page has been designed to inspect the following properties:

- ◆ **Color name offset** Specifies the offset used to display the name of the color. This property is defined by the following value:

```
IlvListGadgetItemHolder::_labelOffsetValue
```

- ◆ **Color rectangle horizontal margin** Specifies the margin between the vertical border of the color rectangle and the border of the gadget item. This property is defined by the following value:

```
IlvColorDrawInfo::_HColorRectMarginValue
```

- ◆ **Color rectangle vertical margin** Specifies the margin between the horizontal border of the color rectangle and the border of the gadget item. This property is defined by the following value:

```
IlvColorDrawInfo::_VColorRectMarginValue
```

- ◆ **Small rectangle** Specifies whether the color rectangle in the gadget item should be drawn in the margin specified by the color name offset property or occupy the whole gadget item. The property is defined by the following value:

```
IlvColorDrawInfo::_SmallColorRectValue
```

- ◆ **Rounded rectangle radius** Specifies the radius applied to the rectangle corners. For the purpose of this example, this property is ignored if the small rectangle property has been set. The property is defined by the following value:

```
IlvColorDrawInfo::_ColorRoundRectRadius
```

- ◆ **Visible items** Specifies which items appear in the drop-down list of the combo box. The property is defined by the following value:

```
IlvScrolledComboBox::_nbVisibleItemsValue
```

- ◆ **Enable large list option** Specifies the “Enable large list” option for the combo box. For details about this option, see the class `IlvScrolledComboBox` in the IBM ILOG Views *Reference Manual*. The property is defined by the following value:

```
IlvScrolledComboBox::_largeListValue
```


To inspect the above properties, you must define the `initializeEditors` member function as follows:

```
void
IlvColorComboBoxInspectorPanel::initializeEditors()
{
    // Color name offset.
    link("ColorNameOffset", IlvListGadgetItemHolder::_labelOffsetValue);

    // Horizontal margin.
    link("XColorMargin", IlvColorDrawInfo::_HColorRectMarginValue);

    // Vertical margin.
    link("YColorMargin", IlvColorDrawInfo::_VColorRectMarginValue);

    // Small rectangle editor.
    link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);

    // Rounded rectangle editor.
    link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);

    // Visible items.
    link("ComboVisibleItems", IlvScrolledComboBox::_nbVisibleItemsValue);
}
```

The `link` method automatically builds an editor and associates it with the gadget whose name is passed as its first parameter. It also creates an accessor to the property provided as its second parameter. It then links the editor and the accessor, which will be used in conjunction to inspect the property. For more information about this function, refer to the class `IlvStInspectorPanel` in the reference manual.

Previewing Changes

The changes made to the color combo box properties via the inspector panel can be reflected in a preview gadget. To achieve this, you have to create an accessor to the gadget that you decide to use as the preview gadget. To create this accessor, we recommend that you use the class `IlvStIGraphicContainerAccessor`. Once this is done, you register the accessor as the preview accessor to the inspected property using the `setPreviewAccessor` or `setPreviewValueAccessor` member functions (`IlvStIEditor` and `IlvStIPropertyAccessor`). Here is what you should do to implement a preview gadget for the properties mentioned earlier in this section:

```
void
IlvColorComboBoxInspectorPanel::initializeEditors()
{
    IlvStIPropertyAccessor* previewGadgetAcc =
        new IlvStIGraphicContainerAccessor(getHolder(), "ColorItemsList");

    IlvStIEditor* editor;

    // Color name offset.
    editor = link("ColorNameOffset",
        IlvListGadgetItemHolder::_labelOffsetValue);
    editor->setPreviewValueAccessor(
```

```

        previewGadgetAcc,
        IlvListGadgetItemHolder::_labelOffsetValue);

// Horizontal margin.
editor = link("XColorMargin", IlvColorDrawInfo::_HColorRectMarginValue);
editor->setPreviewValueAccessor(previewGadgetAcc,
                               IlvColorDrawInfo::_HColorRectMarginValue);

// Vertical margin.
editor = link("YColorMargin", IlvColorDrawInfo::_VColorRectMarginValue);
editor->setPreviewValueAccessor(previewGadgetAcc,
                               IlvColorDrawInfo::_VColorRectMarginValue);

// Small rectangle editor.
editor = link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);
editor->setPreviewValueAccessor(previewGadgetAcc,
                               IlvColorDrawInfo::_SmallColorRectValue);

// Rounded rectangle editor.
editor = link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);
editor->setPreviewValueAccessor(previewGadgetAcc,
                               IlvColorDrawInfo::_ColorRoundRectRadius);
}

```

Using Preconditions

Earlier in this section, we said that the Rounded rectangle radius property is ignored when the small rectangle property is set. The following code shows you how to implement this condition:

```

// Small rectangle editor.
editor = link("SmallRect", IlvColorDrawInfo::_SmallColorRectValue);
IlvStIPropertyAccessor* smallRectAcc =
    (IlvStIPropertyAccessor*) editor->getAccessor();

// Rounded rectangle editor.
editor = link("RoundRadius", IlvColorDrawInfo::_ColorRoundRectRadius);
editor->getAccessor()->setPrecondition(
    new IlvStIPreconditionValue(smallRectAcc,
                                IlFalse, (IlvInt)0));
...

```

Implementing the Items Page

The Items page allows the user to edit the list of colors displayed in the combo box. To handle the list of colors, we must first define a list accessor by deriving the class `IlvStIPropertyListAccessor`, as illustrated below. List accessors are described in the *List Accessors* on page 172.

```

class IlvColorItemsAccessor
: public IlvStIPropertyListAccessor {
public:
    // -----
    // Constructor / destructor
    ....
    // -----
    IlvListGadgetItemHolder* getListGadgetItemHolder() const;

```

```

protected:
    IlvGadgetItem* getGadgetItem(const IlvStIProperty* property) const;

    virtual IlvStIProperty** getInitialProperties(IIUInt& count);
    virtual IlvStIProperty* createDefaultProperty() const;

    virtual void addProperty(IlvStIProperty* property, IIUInt index);
    virtual void replaceProperty(IlvStIProperty* origProperty,
        IlvStIProperty* newProperty,
        IIUInt index);
    virtual void deleteNewProperty(IlvStIProperty* property);
    virtual void deleteProperty(IlvStIProperty* property, IIUInt index);
};

```

The `getListGadgetItemHolder` method returns the gadget item holder that contains the colors to be displayed. This value is the one returned by the `IlvStIPropertyAccessor` passed to the constructor.

```

IlvListGadgetItemHolder*
IlvColorItemsAccessor::getListGadgetItemHolder() const
{
    IlvStIProperty* property = (_accessor? _accessor->get() : 0);
    return (property? (IlvListGadgetItemHolder*)property->getPointer() : 0);
}

```

The `getGadgetItem` method returns the gadget item stored in the property provided as its parameter.

```

IlvGadgetItem*
IlvColorItemsAccessor::getGadgetItem(const IlvStIProperty* property) const
{
    return (IlvGadgetItem*)(property? property->getPointer() : 0);
}

```

The `getInitialProperties` method returns an array of properties which corresponds to the initial colors contained in the combo box.

```

IlvStIProperty**
IlvColorItemsAccessor::getInitialProperties(IIUInt& count)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return 0;

    count = (IIUInt)listHolder->getCardinal();
    if (!count)
        return 0;
    IlvStIProperty** properties = new IlvStIProperty*[count];
    for(IIUInt i = 0; i < count; i++) {
        properties[i] =
            new IlvStIValueProperty(
                (IlvAny)listHolder->getItem((IlvUShort)i), "Item");
    }
    return properties;
}

```

The `createDefaultProperty` method is called when the user presses the Add button to create a new color. By default, this color is black.

```
IlvStIProperty*
IlvColorItemsAccessor::createDefaultProperty() const
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return 0;
    IlvValue valueInfo(IlvColorDrawInfo::_ColorInfosValue->name());
    IlvColorDrawInfo* colorInfo = (IlvColorDrawInfo*)(IlvAny)
        listHolder->getGadget()->queryValue(valueInfo);
    return new IlvStIValueProperty(
        new IlvColorGadgetItem(listHolder->getGadget()->
            getDisplay()->getColor("Black"),
            colorInfo),
        "Item");
}
```

The `addProperty` method is called when changes are applied to add the gadget item contained in the property given as its first parameter to the position specified by the `index` parameter. The gadget item is added to the combo box.

```
void
IlvColorItemsAccessor::addProperty(IlvStIProperty* property, IluInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (listHolder)
        listHolder->insertItem(getGadgetItem(property),
            (IlvShort)(IlvUShort)index);
}
```

The `replaceProperty` method is called when changes are applied to replace the gadget item contained in the property given as its first parameter by the gadget item contained in the property given as its second parameter. The third parameter indicates the position of the replaced gadget item in the combo box.

```
void
IlvColorItemsAccessor::replaceProperty(IlvStIProperty* origProperty,
                                       IlvStIProperty* newProperty,
                                       IluInt)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    IlvGadgetItem* origGadgetItem = getGadgetItem(origProperty);
    IlvGadgetItem* newGadgetItem = getGadgetItem(newProperty);
    *(origGadgetItem) = *newGadgetItem;
    newProperty->setPointer(origGadgetItem);
}
```

The `deleteNewProperty` method deletes the gadget item contained in the property passed as its parameter. This method is invoked when the changes made by the user are cancelled, to destroy the gadget item created by pressing the Add button. Since this gadget item is not actually added to the combo box, it does not have to be removed from it.

```

void
IlvColorItemsAccessor::deleteNewProperty(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem(property);
    if (gadgetItem)
        delete gadgetItem;
}

```

The `deleteProperty` method is called when changes are applied to remove the gadget item contained in the property given as the parameter from the color combo box.

```

void
IlvColorItemsAccessor::deleteProperty(IlvStIProperty*, IUInt index)
{
    IlvListGadgetItemHolder* listHolder = getListGadgetItemHolder();
    if (!listHolder)
        return;
    listHolder->removeItem((IlvShort)(IlvUShort)index);
}

```

Reusing the Color List Accessor

As we have seen throughout the example, the list accessor does not access the combo box directly but through its gadget item holder. The same list accessor can therefore be reused to inspect a color string list. To access the gadget item holder of the inspected combo box, we just have to create a combined accessor, as shown in the following code sample. For a description of combined accessors, see the section *Combined Accessors* on page 172. This combined accessor will be provided as a parameter to the `IlvColorItemsAccessor` constructor.

```

class IlvColorGadgetItemHolderAccessor
: public IlvStICombinedAccessor
{
public:
    IlvColorGadgetItemHolderAccessor(IlvStIPropertyAccessor* accessor = 0,
                                     UpdateMode updateMode = NoUpdate,
                                     BuildMode buildMode = None,
                                     const char* name = 0);

    // -----
protected:
    virtual IlvStIProperty* getOriginalValue();
};

IlvStIProperty*
IlvColorGadgetItemHolderAccessor::getOriginalValue()
{
    IlvStIProperty* property =
        (getObjectAccessor()? getObjectAccessor()->get() : 0);
    if (!property)
        return 0;
    IlvColorComboBox* combo = (IlvColorComboBox*)property->getPointer();
    if ((!combo) || (!combo->getStringList()))
        return 0;
    return new IlvStIValueProperty((IlvListGadgetItemHolder*)combo,
                                   "ColorHolder");
}

```

The class `IlvColorItemsAccessor` is used in the inspector panel in the following way:

```
IlvColorItemsAccessor* lstAccessor =
    new IlvColorItemsAccessor(
        new IlvColorGadgetItemHolderAccessor(getInspectedGraphicAccessor()));
```

Modifying Colors in the List

In the previous section, we explained how to access a list of colors. We are now going to modify a color selected in the list. To do so, we define a class that lets us inspect the color of an `IlvColorGadgetItem` gadget item. Since the color is defined by the gadget item label, changing the label implies changing the color.

```
class IlvGadgetItemColorAccessor
: public IlvStICombinedAccessor
{
public:
    ...
    // -----
protected:
    IlvGadgetItem* getGadgetItem() const;
    virtual IlvStIProperty* getOriginalValue();
    virtual void applyValue(IlvStIProperty*);
};

IlvGadgetItem*
IlvGadgetItemColorAccessor::getGadgetItem() const
{
    IlvStIProperty* property =
        (getObjectAccessor()? getObjectAccessor()->get() : 0);
    return (property? (IlvGadgetItem*)property->getPointer() : 0);
}

IlvStIProperty*
IlvGadgetItemColorAccessor::getOriginalValue()
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    if (!gadgetItem)
        return 0;
    return new IlvStIValueProperty(gadgetItem->getLabel(), "Color");
}

void
IlvGadgetItemColorAccessor::applyValue(IlvStIProperty* property)
{
    IlvGadgetItem* gadgetItem = getGadgetItem();
    if (!gadgetItem)
        return;
    IlvValue value;
    gadgetItem->setLabel((const char*)property->getValue(value));
}
```

This class is used in the inspector panel code as follows:

```
editor = new IlvStIPropertyColorEditor("EditColorItem",
    new IlvGadgetItemColorAccessor(lstAccessor->getSelectionAccessor()));
addEditor(editor);
```

The class `IlvStIPropertyColorEditor` interfaces with a selection field to make it possible to select a color. The label that appears in the selection field represents the name of the selected color.

Creating the Color List Editor

To display a list of items, it is common practice to use an `IlvStringList`, which is handled by an `IlvStIPropertyListEditor`. However in the case of our example, we want the list to display color gadget items instead of character strings. We have implemented such list using the class `IlvColorStringList`, a subclass of `IlvStringList`. To interface this new class, we have defined the following editor class:

```
class IlvColorListEditor
: public IlvStIPropertyListEditor {
public:
    // -----
    // Constructor / destructor
    ...
    // -----
    // Overridables.
    virtual IlBoolean connectHolder(IlvGraphicHolder* holder);
protected:
    virtual IlvGadgetItem* createGadgetItem(
                                const IlvStIProperty* property) const;
};

IlBoolean
IlvColorListEditor::connectHolder(IlvGraphicHolder* holder)
{
    // Replaces string list of colors by an IlvColorStringList.
    IlvGraphicHolder* subHolder;
    IlvGadget* oldList =
        (IlvGadget*)IlvStIFindGraphic(holder, getName(), &subHolder);
    if (!oldList)
        return IlvStIPropertyListEditor::connectHolder(holder);
    IlvRect bbox;
    oldList->boundingBox(bbox);
    IlvColorStringList* colorList =
        new IlvColorStringList(oldList->getDisplay(),
                               bbox,
                               oldList->getThickness(),
                               oldList->getPalette());
    colorList->useFullSelection(IlTrue, IlFalse);
    colorList->setSelectionMode(IlvStringListSingleSelection);
    colorList->setExclusive(IlTrue);
    colorList->scrollBarShowAsNeeded(IlTrue, IlTrue, IlFalse);
    subHolder->getContainer()->replace(oldList, colorList, IlTrue);

    return IlvStIPropertyListEditor::connectHolder(holder);
}

IlvGadgetItem*
IlvColorListEditor::createGadgetItem(const IlvStIProperty* property) const
{
    IlvGadgetItem* gadgetItem = (IlvGadgetItem*)property->getPointer();
    if (!gadgetItem)
        return 0;
}
```

```

IlvValue valueInfo(IlvColorDrawInfo::_ColorInfosValue->name());
IlvColorDrawInfo* colorInfo =
    (IlvColorDrawInfo*)(IlvAny)getListGadget()->queryValue(valueInfo);
IlvGadgetItem* newGadgetItem =
    new IlvColorGadgetItem(getDisplay()->getColor(gadgetItem-
>getLabel()),
                           colorInfo);
newGadgetItem->setEditable(IfFalse);
return newGadgetItem;
}

```

Declaring Accessors and Editors for Inspecting Color Items to the Inspector Panel

Accessors and editors for inspecting color items are declared in the `initializeEditors` method as follows:

```

IlvColorItemsAccessor* lstAccessor =
    new IlvColorItemsAccessor(
        new IlvColorGadgetItemHolderAccessor(getInspectedGraphicAccessor()));
addEditor(new IlvColorListEditor(lstAccessor, "ColorItemsList"));

IlvStIEditor* editor =
    new IlvStIPropertyColorEditor("EditColorItem",
        new IlvGadgetItemColorAccessor(lstAccessor->getSelectionAccessor()));
addEditor(editor);

```


Part II

IBM ILOG Views Gadgets

This part provides information for developing applications that incorporate IBM® ILOG® Views Gadgets. It contains the following chapters:

- ◆ **Chapter 8, *Introducing IBM ILOG Views Gadgets*** provides an overview of the IBM ILOG Views Gadgets package.
- ◆ **Chapter 9, *Understanding Gadgets*** describes the holder objects, properties, and predefined look and feel features available with IBM ILOG Views gadgets.
- ◆ **Chapter 10, *Dialogs*** introduces the classes for creating dialogs.
- ◆ **Chapter 11, *Using Common Gadgets*** introduces the common gadget classes and explains how to use them.
- ◆ **Chapter 12, *Gadget Items*** introduces gadget items and explains how to use them.
- ◆ **Chapter 13, *Menus, Menu Bars, and Toolbars*** introduces the classes for creating menus and tool bars and explains how to use them.
- ◆ **Chapter 14, *Matrices*** introduces the classes for creating matrices and explains how to use them.

- ◆ **Chapter 15, *Panes*** explains how to create graphical user interfaces that incorporate panes.
- ◆ **Chapter 16, *Docking Panes and Containers*** explains how to create graphical user interfaces that feature docking panes.
- ◆ **Chapter 17, *View Frames*** explains how to create graphical user interfaces that feature docking panes.
- ◆ **Chapter 18, *Customizing the Look and Feel*** introduces the classes for creating look and feel and explains how to customize and create your own look and feel mechanism.

Introducing IBM ILOG Views Gadgets

The IBM® ILOG® Views Gadgets package is a C++ class library for building interactive graphical user interfaces. This package is built on top of the IBM ILOG Views Foundation package and is composed of classes for creating special graphic objects, called gadgets, which you can add to container objects to create graphic panels or interfaces. Buttons, tool bars, and menus are some of the many interactive graphic objects you can create with IBM ILOG Views Gadgets.

This introductory chapter contains the following:

- ◆ *Gadgets Main Features*
- ◆ *Gadgets Libraries*
- ◆ *Gadgets in a Snapshot*

Gadgets Main Features

The IBM® ILOG® Views Gadgets library provides:

- ◆ A large set of lightweight graphic objects, such as buttons, text fields, menus, and toolbars.
- ◆ A large set of gadget containers, including several predefined dialog boxes.

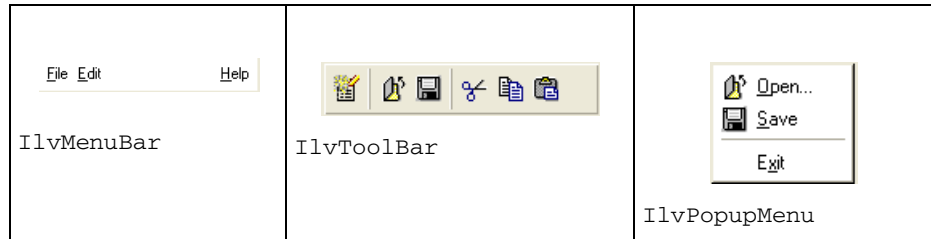
- ◆ Four predefined presentation styles: Motif®, Microsoft® Windows® 3.11, Microsoft Windows 95, and Microsoft Windows XP.
- ◆ An easy way to create your own presentation style.
- ◆ A library that is portable to UNIX® workstations and PCs running Microsoft Windows.
- ◆ An easy way to combine applications written with a standard widget toolkit, such as Motif and Microsoft Windows, with new applications using IBM ILOG Views gadgets.

Gadgets in a Snapshot

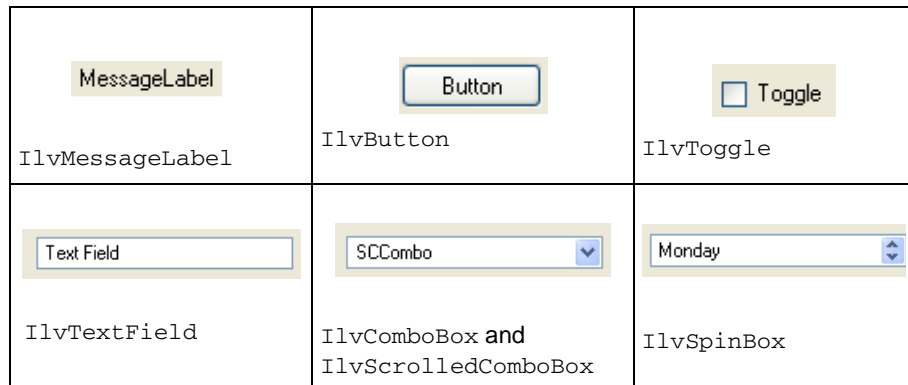
The base class for all the gadgets is `IlvGadget`. This class derives from `IlvGraphic`, a class of the IBM® ILOG® Views Foundation library.



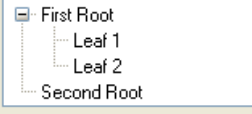

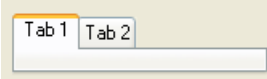
The following illustrations show the various gadgets that the Gadgets library provides:

Menus

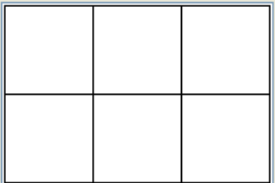
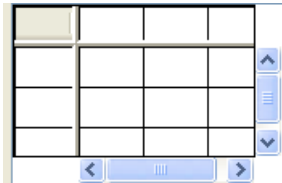
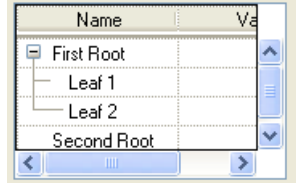


Common Gadgets



 <p>IlvStringList</p>	 <p>IlvText</p>	 <p>IlvTreeGadget</p>
 <p>IlvFrame</p>	 <p>IlvNotebook</p>	

Matrices

 <p>IlvMatrix</p>	 <p>IlvSheet</p>	 <p>IlvHierarchicalSheet</p>
--	---	--

Gadgets Libraries

For each gadget that you want to use in your application you have to include the appropriate header file. Header files for gadgets can be found in the following directory:

```
ILVHOME/include/ilviews/gadgets
```

You must also link your application with the following gadget library:

- ◆ `ilvgadgt.lib` for Microsoft® Windows® platforms
- ◆ `libilvgadgt` for UNIX® platforms

If you are using advanced gadgets, you must link your application with the following gadget library:

- ◆ `ilvadvgt.lib` for Microsoft Windows platforms
- ◆ `libilvadvgt` for UNIX platforms

To know whether a gadget class is located in the standard or advanced gadget library, refer to the Reference Manual.

Note: *The gadget libraries use resources that are located under the `ILVHOME` directory. If you do not want to set `ILVHOME`, or if IBM ILOG Views is not installed on the target computer, you must add those resources to your application.*

You must also link your application with the look-and-feel gadgets libraries, depending on the look and feel your application will use. By default, an application ran on UNIX will use the Motif® look, and an application ran on Windows will use one of the provided Windows looks. See the section *Gadgets Look and Feel* on page 217 for details.

Code Sample

Here is a very basic program that displays a container with a button. Clicking on the button exits the program.

```
#include <ilviews/gadgets/button.h>
#include <ilviews/gadgets/gadcont.h>

void Quit(IlvGraphic*, ILAny arg)
{
    IlvDisplay* display = (IlvDisplay*)arg;
    delete display;
    IlvExit(0);
}

int main(int argc, char* argv[])
{
    // Create the display.
    IlvDisplay* display = new IlvDisplay("Hello", "", argc, argv);
    if(!display)
        return 0;
    if(display->isBad()){
        delete display;
        return 1;
    }

    // Create the container.
    IlvGadgetContainer* cont =
        new IlvGadgetContainer(display, "Hello", "Hello", IlvRect(0,0,100,100));
    cont->moveToScreen(IlvCenter);

    // Add the button.
    IlvButton* button = new IlvButton(display, IlvPoint(30, 30), "Click Me !");
    button->addCallback(Quit, display);
    cont->addObject(button);

    // Show the container and run the event loop.
    cont->show();
    IlvMainLoop();

    return 0;
}
```


Understanding Gadgets

This chapter introduces you to properties that are common to all the gadgets in the library. It covers the following topics:

- ◆ *Gadget Holders*
- ◆ *Common Gadget Properties*
- ◆ *Gadgets Look and Feel*

Gadget Holders

Gadget holders are objects for storing, displaying, and handling gadgets. The main class for gadget holders is the `IlvGadgetContainer` class. This class derives from `IlvContainer` and thus inherits from all the features this superclass provides, such as member functions for adding or removing objects. It also provides basic features such as keyboard focus management, attachments, and tooltips.

In this section, you will find information on the following topics:

- ◆ *List of Available Gadget Holders*
- ◆ *Handling Events*
- ◆ *Focus Management*

◆ *Gadgets Attachments*

List of Available Gadget Holders

The `IlvGadgetContainer` objects are not the only gadget holders that IBM® ILOG® Views Gadgets provide. This section introduces you to other available gadget holders:

- ◆ *Gadget Managers*
- ◆ *Notebooks*
- ◆ *Matrices*
- ◆ *Toolbars*
- ◆ *Paned Containers*

You will find also information about:

- ◆ *Limitations* in the use of gadget holders

Gadget Managers

The `IlvGadgetManager` class is a subclass of the `IlvManager` class that deals with gadgets. For details about managers, see the related *User's Manual*. Unlike `IlvManager` objects, instances of `IlvGadgetManager` have only one associated view because gadgets, cannot appear in several views at the same time, whereas basic graphic objects can.

As a general rule, unless you want to save gadgets to an `.ilv` file (the IBM ILOG Views format), we recommend that you use gadget containers rather than gadget managers to store gadgets.

Notebooks

You can display gadgets inside notebook pages. Actually, default notebook pages are implemented using gadget containers. For more information, see *Handling Notebook Pages* on page 244.

Matrices

A matrix is a special gadget made up of rows and columns. Each matrix item can contain a gadget that has its own behavior inside the matrix. For details, see *Using Gadgets in a Matrix* on page 315.

Toolbars

Gadgets can be displayed inside a toolbar. For details, see *Managing Gadgets in a Toolbar* on page 300.

Paned Containers

The `IlvPanedContainer` class is a subclass of `IlvGadgetContainer` that divides the container into panes. Each panes can encapsulate a gadget. For details, see *Creating a Graphic Pane* on page 323.

Limitations

We do not recommend that you use simple containers to store gadgets since these objects do not implement features such as keyboard focus management. Adding gadgets to these containers might produce unexpected results.

In addition, gadgets cannot be zoomed in or out. As a consequence, we do not recommend that you modify the scaling factor of a gadget holder.

Handling Events

Gadget holders are responsible for dispatching events to the gadgets. The `IlvGadget` class has a `handleEvent` member function that processes user events, such as clicking the mouse or using the keyboard. Unlike with other graphic objects, you do not have to set an interactor to a gadget to be able to use it.

Note: *However, you can set an interactor to a gadget if you want to.*

The `handleEvent` member function is virtual and can be redefined in subclasses to handle additional events.

Gadget Holder Events

When the mouse enters or leaves an `IlvGadget` object, its associated gadget holder generates the `IlvMouseEnter` and `IlvMouseLeave` events (These two events are defined in the enum `IlvEventType`). These events are sent to the gadget, or to its associated interactor, if any, and are processed by the `handleEvent` member function. Then, the virtual member functions `IlvGadget::enterGadget` or `IlvGadget::leaveGadget` are called. By default, these member functions invoke the `Enter Gadget` and the `Leave Gadget` callbacks, respectively. See *Associating a Callback with a Gadget* on page 210.

One consequence of this is that you cannot have the `IlvMouseEnter` and `IlvMouseLeave` event trigger an accelerator because an accelerator is attached to an `IlvView` object. The `IlvView` object does not have knowledge of these events.

Note: *This only applies to the `IlvGadgetContainer` and `IlvGadgetManager` classes.*

Focus Management

Gadget holders manage the keyboard focus. For a gadget, having the focus means that it can receive a keyboard event. A gadget has the focus when the user clicks on it with the mouse.

Pressing the Tab key moves the focus to the next gadget. Pressing Shift-Tab moves it to the previous gadget. By default, the Tab key moves from one gadget to another first from left to right, then from top to bottom. However, you can specify the order in which the gadgets are given the focus when the Tab key is pressed by defining a focus chain.

This section explains the following:

- ◆ *Defining a Focus Chain*
- ◆ *Setting a Focus Chain Between Gadget Holders*
- ◆ *Notifying a Change of Focus*

Defining a Focus Chain

Only the gadgets stored in the same gadget holder can be linked by a focus chain. To define a focus chain, use the following member functions of the `IlvGraphic` class:

- ◆ `IlvGraphic::setNextFocusGraphic`
- ◆ `IlvGraphic::setPreviousFocusGraphic`
- ◆ `IlvGraphic::setLastFocusGraphic`
- ◆ `IlvGraphic::setFirstFocusGraphic`

The name parameter provided in the member functions `setNextFocusGraphic` and `setPreviousFocusGraphic` is a symbolic name that must be created from the name of the target gadget. For example, if you want the next object in the focus chain of `gadget` to be the gadget named “Button”, call:

```
gadget->setNextFocusGraphic (IlvGetSymbol ("Button")) ;
```

Setting a Focus Chain Between Gadget Holders

By default, the focus loops back to the first gadget in the chain when the user reaches the last gadget in the focus chain. You can, however, force the focus to another gadget holder that you specify using the following member functions:

- ◆ `IlvGraphicHolder::getNextFocusHolder`
- ◆ `IlvGraphicHolder::setNextFocusHolder`
- ◆ `IlvGraphicHolder::getPreviousFocusHolder`

Notifying a Change of Focus

When the keyboard focus enters or leaves an `IlvGadget` object, its associated gadget holder generates the `IlvKeyboardFocusIn` and `IlvKeyboardFocusOut` events. These events are sent to the gadget, or to its associated interactor, if any, and are processed by the `handleEvent` member function. Then, the virtual member functions `IlvGadget::focusIn` and `IlvGadget::focusOut` are called. By default these member

functions invoke the Focus In and Focus Out callbacks, respectively. See *Associating a Callback with a Gadget* on page 210.

Gadgets Attachments

The gadget holder provides an attachment model that manages the geometry of the gadgets when the holder is resized. This attachment model is defined by the `IlvGraphicHolder` interface.

You can get a pointer to the `IlvGraphicHolder` interface using the `getHolder` member function.

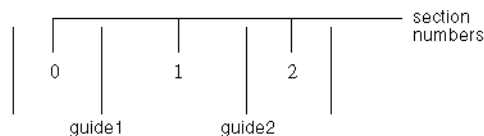
To attach a gadget to its gadget holder, you have to define guides.

This section covers the following topics:

- ◆ *Introducing Guides and Sections*
- ◆ *Attaching a Gadget to Guides*
- ◆ *Setting the Weight of a Gadget*

Introducing Guides and Sections

Guides split gadget holders into several sections, either horizontally or vertically:



Guides are not numbered, whereas the sections they delimit are. When a new guide is added, sections are renumbered to include the resulting new sections.

By default, there are no guides, except those alongside the window borders.

When the holder is resized, each of its sections are resized according to their weight. The weight of a section is the portion of the window that is allocated to it (delimited by the guide) relative to other sections, when the window is resized. The following formula is applied to each section when the window is resized:

$$\text{New section size} = \text{Initial section size} + \text{Delta} \times \frac{\text{Weight of guide delimiting the section}}{\text{Sum of weights of all guides}}$$

where Delta equals the new size of the window minus its initial size.

It is possible to manipulate guides using the `IlvGraphicHolder` member functions listed below:

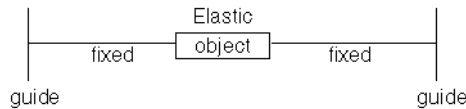
- ◆ `addGuide`
- ◆ `removeGuide`
- ◆ `getGuideCardinal`
- ◆ `getGuidePosition`
- ◆ `getGuideSize`
- ◆ `getGuideWeight`
- ◆ `getGuideLimit`

Attaching a Gadget to Guides

Once guides have been defined, it is possible to attach a gadget to them using the member function `IlvGraphicHolder::attach`:

```
holder->attach(object);
```

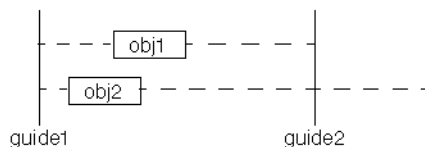
This code attaches gadgets to the guides as shown in the following diagram:



In this example, we use the default guides located along the window borders. However, you can use the last three parameters of the `attach` member function to specify other guides.

```
holder->attach(obj1, IlvHorizontal, 0, 1, 0, 1, 1);  
holder->attach(obj2, IlvHorizontal, 0, 1, 0, 1, 2);
```

This will produce the following result:

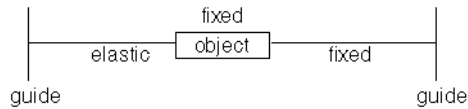


Setting the Weight of a Gadget

The third, fourth, and fifth parameters of the `attach` member function define the weight before the gadget, the gadget weight, and the weight after the gadget, respectively. These weights are used in the same manner as the guides weight, using the same formula. For example, the following call:

```
holder->attach(object, IlvHorizontal, 1, 0, 0);
```

will result in the following attachments:



Common Gadget Properties

In this section, you will find information on the following topics:

- ◆ *Gadget Appearance*
- ◆ *Associating a Callback with a Gadget*
- ◆ *Localizing a Gadget*
- ◆ *Associating a Mnemonic with a Gadget Label*
- ◆ *Setting Tooltips*
- ◆ *Gadget Resources*

Gadget Appearance

You can define the appearance of a gadget by:

- ◆ *Setting a Gadget as Sensitive*
- ◆ *Setting the Thickness of a Gadget*
- ◆ *Setting a Gadget as Transparent*
- ◆ *Showing or Hiding the Gadget Frame*

Setting a Gadget as Sensitive

A gadget is said to be sensitive if it responds to events, that is, if something happens when the user clicks on it. The visual appearance of sensitive gadgets is different from that of nonsensitive ones, as shown in the illustration below:

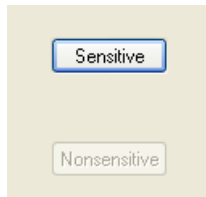


Figure 9.1 Sensitive Gadget versus Nonsensitive Gadget

To change the sensitivity of a gadget, use the member function `IlvGraphic::setSensitive`.

You can also use the member function `IlvGadget::setActive` with `IlFalse` as parameter to specify that a gadget should not respond to events.

The difference between this method and the `setSensitivity` member function is that the drawing of the gadget does not change and the `handleEvent` member function of the gadget is not called.

Setting the Thickness of a Gadget

You can customize the appearance of a gadget by modifying its thickness. The thickness defines the size of the shadow that is used to draw borders, decorations, and so on. To change the thickness of a gadget, use the member function `IlvGadget::setThickness`.

Note: Depending on the look and feel in use, modifying the thickness of a gadget may not affect the way it appears. More specifically, on Microsoft® Windows® and Microsoft Windows 95, most of the gadgets do not take thickness into account.

The following illustration shows two buttons with different thicknesses in the Motif look:



Figure 9.2 Buttons with Different Thickness Values

Setting a Gadget as Transparent

By default, all gadgets are opaque except for message labels, which are transparent. See *Using `IlvMessageLabel`* on page 239. You can make a gadget transparent by calling the member function `IlvGadget::setTransparent` with `IlTrue` as its parameter. All the gadgets in the following illustration are transparent. The transparent setting allows for the background texture to show through. The gadgets are shown here with the Windows 95 look-and-feel:

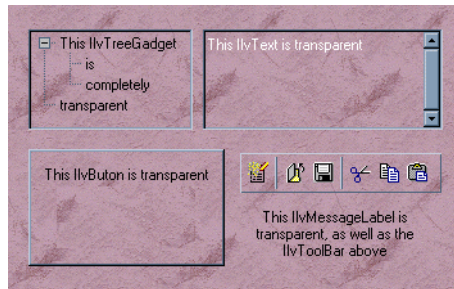
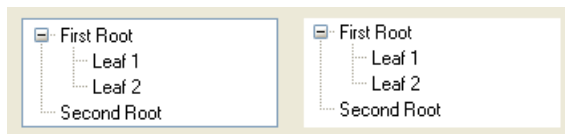


Figure 9.3 Transparent Gadgets

Note: The scrolling of scrollable gadgets may be slowed down when these gadgets are set to transparent.

Showing or Hiding the Gadget Frame

Most of the gadgets use a frame to give a relief aspect. The frame of a gadget is the last part of the gadget to be drawn. It is drawn by calling the `IlvGadget::drawFrame` member function. You can choose to change the frame visibility by calling the `IlvGadget::showFrame` member function. The following figure shows two gadgets, one with a frame, and the other one without:



Associating a Callback with a Gadget

You can associate a callback function with a gadget using the method `IlvGraphic::addCallback`. A callback function is generally invoked by the `handleEvent` member function of its associated gadget when the user performs an action on it. The prototype of the callback function is defined by the `IlvGraphicCallback` type.

A gadget can define several callback types, each one corresponding to a specific user action. Each callback type stores a list of callback functions that will be invoked when the related action is performed.

Predefined Callback Types

Gadgets have predefined callback types:

- ◆ **Main**—This callback type defines the callbacks that carry out the main action attached to a gadget.

- ◆ Focus In—This callback type defines callbacks that carry out actions performed when the gadget is given the focus. The symbol corresponding to this callback type can be retrieved by `IlvGadget::FocusInSymbol`. See *Focus Management* on page 204.
- ◆ Focus Out—This callback type defines callbacks that carry out actions performed when the gadget loses the focus. The symbol corresponding to this callback type can be retrieved by `IlvGadget::FocusOutSymbol`. See *Focus Management* on page 204.
- ◆ Enter Gadget—This callback type defines callbacks that carry out actions performed when the mouse enters the gadget. The symbol corresponding to this callback type can be retrieved by `IlvGadget::EnterGadgetSymbol`. See *Gadget Holder Events* on page 204.
- ◆ Leave Gadget—This callback type defines callbacks that carry out actions performed when the mouse leaves the gadget. The symbol corresponding to this callback type can be retrieved by `IlvGadget::LeaveGadgetSymbol`. See *Gadget Holder Events* on page 204.

For example, if you want to add a Focus In callback to a gadget, you can code:

```
gadget->addCallback(IlvGadget::FocusInSymbol(), callback);
```

where `callback` has been declared as follows:

```
void callback(IlvGraphic* g, IlAny arg) { ... }
```

In addition to these general predefined callback types, each gadget type has specific predefined callback types attached. For more details, see the sections describing the individual gadgets.

Localizing a Gadget

Gadgets containing text can be localized. Localizing a gadget means adapting its text to the language used in the final application. This property allows you to create multilingual applications whose current language can be changed dynamically very easily.

IBM® ILOG® Views lets you create message databases as files where you can store all the text that will be displayed in your final application with its translation to as many languages as you want. The message database file have a `.dbm` extension. See “`IlvMessageDatabase`” in the chapter “Internationalization” of the IBM ILOG Views *Foundation User’s Manual*.

To have the text of a gadget change dynamically depending on the language used in the final application, you must provide a reference to the message database where the text is stored instead of hard-coding it.

Let’s suppose that you have created the following message database:

```
Message: &MenuPrinterSetup
en_US: Printer Setup
fr_Fr: Configuration imprimante
```

You can assign a gadget label the string defined by `&MenuPrinterSetup` like this:

```
gadget->setLabel("&MenuPrinterSetup");
```

Calling the `getLabel` member function will return the string `&MenuPrinterSetup` and invoking the `getMessage` member function will provide the translation to the current language (for example, “Printer Setup” for English and “Configuration imprimante” for French).

Associating a Mnemonic with a Gadget Label

A gadget can be associated with a mnemonic. A mnemonic is an underlined letter in a gadget label that you can use as a keyboard shortcut to activate the gadget.

To associate a mnemonic with a gadget label, type a caret (^) before the letter that you want to use as a mnemonic:

```
gadget->setLabel("^File");
char mnemo = gadget->getMnemonic();
```

Note: To type a caret (^) inside a gadget label, use the Escape sequence: `\^`.

You can have a different mnemonic depending on the language you use. You could, for example, have a special entry in your language database (`.dbm` file) such as:

```
Message: &MenuPrinterSetup
en_US: Printer ^Setup
en_Fr: ^Configuration imprimante
```

In French, the letter used as the mnemonic is “C” whereas it is “S” in English.

Setting Tooltips

Gadgets can be associated with a tooltip. A tooltip is short explanatory text that is displayed when the user places the mouse pointer over a gadget. By default, tooltips are supported by the gadget holders. If you want to use tooltips outside gadget holders, use the class `IlvToolTipHandler`.

This section covers the following topics:

- ◆ *Creating a Tooltip*
- ◆ *Attaching a Tooltip to a Gadget*
- ◆ *Enabling and Disabling Tooltips*
- ◆ *Specific Tooltips*

Creating a Tooltip

A tooltip is an instance of the `IlvToolTip` class. To create a tooltip, call:

```
IlvToolTip* tooltip = new IlvToolTip("This is a test");
```

Attaching a Tooltip to a Gadget

You can attach a tooltip to a gadget with the member function

`IlvGraphic::setNamedProperty` since `IlvToolTip` is a subclass of the `IlvNamedProperty` class:

```
gadget->setNamedProperty(new IlvToolTip("This is a test"));
```

Enabling and Disabling Tooltips

You can enable or disable tooltip management at the application level using the static member function `IlvToolTip::Enable`.

Specific Tooltips

Some gadgets have their own tooltip mechanism, including `IlvToolBar`, `IlvTreeGadget`, `IlvMatrix`, `IlvStringList`, and `IlvPopupMenu`.

For more information, refer to the sections dedicated to these gadgets.

Gadget Resources

The system resource mechanism allows you to customize graphic objects at runtime. Object resources are resolved when an object is added to a gadget holder using the member function `addObject`.

One resource setting can be applicable to an individual object or to an object class. Its scope can also be restricted to an individual *storage* object or to a *storage* class, where *storage* stands for `IlvGadgetContainer` or `IlvGadgetManager`.

Each graphic object class can define a set of significant parameters as resources.

Predefined Object Resources

`IlvGraphic` implements the following object resources:

Resource Name	Description	Value
x	x position	integer string
y	y position	integer string
w or width	horizontal size	integer string
h or height	vertical size	integer string

`IlvSimpleGraphic` implements the following resources.

Resource Name	Description	Value
background	palette background color	color name
foreground	palette foreground color	color name
font	palette font	font name
pattern	palette pattern	pattern name
colorPattern	palette color pattern	pattern name
lineStyle	palette line style	line style name
lineWidth	palette line width	integer string
fillStyle	palette fill style	FillPattern FillMaskPattern FillColorPattern
arcMode	palette arc mode	ArcPie ArcChord
fillRule	palette fill rule	EvenOddRule WindingRule
alpha	palette alpha value	Integer string
antialiasingMode	palette antialiasing mode	DefaultAntialiasing UseAntialiasing NoAntialiasing

Warning: *IlvSimpleGraphic resources are only applied to graphic objects that have the default palette.*

Setting Object Resources

The user defines values for these resources in the same way as described in *Display System Resources: getResource* in Graphic Resources. Even though the syntax is system-dependent, the global structure of a resource setting is the same. The structure is `key value`. The left part of the resource specification, `key`, is more complex than the resource specification described in *Display System Resources: getResource* so that the objects affected by this setting can be easily identified. The key specification is defined as follows:

```
Program.Storage.GraphicObject.Resource
```

Here is the description of these four fields:

- ◆ `Program` can be either an application name or the string `IlogViews`.

- ◆ Storage can be either the name of a gadget container or gadget manager, or the string `IlvGadgetContainer` or `IlvGadgetManager`.
- ◆ `GraphicObject` can be the name of a graphic object (as returned by `IlvGraphic::getName`) or the name of a graphic object class (as returned by `IlvGraphic::className`).
- ◆ Resource is the name of the object resource as it appears in the documentation of the class defining this resource.

The fields `Program`, `Storage` and `GraphicObject` can be replaced by the wild card `'*'`.

It is the responsibility of the application developer to document the names of objects, gadget containers, and gadget managers.

It is the responsibility of the graphic object class designer to document the name of the resources defined by this class.

Example: Specifying Object Resources

Here is how to specify that all instances of the `IlvPolygon` class must be red and filled using the even-odd rule:

- ◆ On X Window, add the following to your `~/ .Xdefaults` file:
 - `IlogViews*IlvPolygon.foreground: red`
 - `IlogViews*IlvPolygon.fillRule: EvenOddRule`
- ◆ On Microsoft Windows, add the following to any `.INI` file:

Section `[IlogViews]` or `[<ApplicationName>]`:

 - `*IlvPolygon.foreground=red`
 - `*IlvPolygon.fillRule=EvenOddRule`

Priorities and Conflicts

When several resource settings are applicable to the same target(s), IBM ILOG Views gives priority to the most precise setting, which means that:

- ◆ any string has priority over `'*'`,
- ◆ an application name has priority over `IlogViews`,
- ◆ a gadget container or gadget manager name has priority over `IlvGadgetContainer` or `IlvGadgetManager`,
- ◆ an object name has priority over an object class.

If a conflict remains in spite of these priorities, the result is undefined.

Example: Resource Priority

Using X Window syntax:

1. IlogViews.*.*.foreground: blue
2. myApp.*.*.foreground: green
3. myApp.*.IlvButton.foreground: red
4. myApp.myPanel.*.foreground: yellow
5. myApp.myPanel.myButton.foreground: cyan

Line 5 has priority over all the others.

Line 4 has priority over lines 1 and 2.

There is an unresolved conflict between lines 3 and 4. The color of an `IlvButton` in a gadget container called `myPanel` is not predictable.

Adding New Resources

If you want to add a new resource to a graphic object, you have to overload the virtual member function `IlvGraphic::applyResources`. This method loads object resources and is called by the `addObject` member function of `IlvGadgetContainer` and `IlvGadgetManager`.

When overloading this method, subclasses should call the `applyResources` method of the superclass, then they should use the second `IlvDisplay::getResource` member function to fetch possible values for the new resources they define:

```
const char* getResource(const char* resourceName,
                       const char* objectName,
                       const char* objectClassName,
                       const char* storageName = 0,
                       const char* storageClassName = 0) const;
```


Example: Adding Resources

```
// Assuming class MyObjectClass: public MyObjectSuperClass
// defining a method setLabel.
// The following defines a resource called "labelString".

void MyObjectClass::applyResources(const char* storageName,
                                   const char* storageClassName,
                                   const char* objectName,
                                   const char* objectClassName,
                                   IlvDisplay* display)
{
    if (!display)
        display = getDisplay();
    MyObjectSuperClass::applyResources(storageName,
                                       storageClassName,
                                       objectName,
                                       objectClassName,
                                       display);
    const char* resource = display->getResource("labelString",
                                               objectName,
                                               objectClassName,
                                               storageName,
                                               storageClassName);

    if (resource)
        setLabel(resource);
}
```

Gadgets Look and Feel

The appearance and behavior of some gadgets can be modified to conform to the graphic environment in which they are being used. Currently, IBM® ILOG® Views takes into account four graphic environments: Motif®, Microsoft® Windows® 3.11, Microsoft Windows 95, and Microsoft Windows XP. You can decide which look and feel is to be used by your gadgets. You can also define a custom look and feel by inheriting an existing one, or by completely redesigning your own look and feel.

In this section, you will find information on the following topics:

- ◆ *Using the Default Look and Feel*
- ◆ *Using Several Look and Feel*
- ◆ *Dynamic Loading of Look and Feel*
- ◆ *Changing the Look and Feel Dynamically*
- ◆ *Using the Windows XP Look and Feel*

Using the Default Look and Feel

By default, only one style is used in a given IBM® ILOG® Views program, which is the standard style of the computer system:

- ◆ Motif® style on UNIX®
- ◆ Microsoft® Windows 3®.11 style on Windows 3.x and Microsoft Windows NT 3.x
- ◆ Microsoft Windows 95 style on Windows 95 Windows NT 4, and Windows 2000
- ◆ Microsoft Windows XP style on Windows XP

Note: You can override this default setting by using the `ILVLOOK` environment variable, or the `LOOK` resource. In this case, be sure to provide access to the specified look to your application, or it will not be used. See the section *Using Several Look and Feel on page 219* and *Dynamic Loading of Look and Feel on page 220*.

Note: The Microsoft Windows XP style is only available on computers running Microsoft Windows XP. An IBM ILOG Views application built on a Microsoft Windows XP platform may not run on a previous version of Microsoft Windows (Windows 2000, NT, and so on). For more details, see the section *Using the Windows XP Look and Feel on page 222*.

Depending on the platform on which you are building your application, it must be linked with the corresponding look-and-feel libraries. The following tables sum up the different libraries available:

Table 9.1 Look Libraries for Windows Platforms

Look	Standard Gadgets Library	Advanced Gadgets Library
Motif	<code>ilvmllook.lib</code>	<code>ilvamlook.lib</code>
Windows 3.11	<code>ilvwlook.lib</code>	<code>ilvawlook.lib</code>
Windows 95	<code>ilvw95look.lib</code> , <code>ilvwlook.lib</code>	<code>ilvaw95look.lib</code> , <code>ilvawlook.lib</code>
Windows XP	<code>ilvwxplook.lib</code> , <code>ilvw95look.lib</code> , <code>ilvwlook.lib</code> , <code>uxtheme.lib</code>	<code>ilvawxplook.lib</code> , <code>ilvaw95look.lib</code> , <code>ilvawlook.lib</code>

Note that the `uxtheme.lib` is a Microsoft library. If this library is not present on your computer, it is available in the Microsoft Platform SDK. To get the SDK, go to <http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en>.

Table 9.2 Look Libraries for UNIX Platforms

Look	Standard Gadgets Library	Advanced Gadgets Library
Motif	<code>libilvmlook</code>	<code>libilvamlook</code>
Windows 3.11	<code>libilvwlook</code>	<code>libilvawlook</code>
Windows 95	<code>libilvw95look</code> , <code>libilvwlook</code>	<code>libilvaw95look</code> , <code>libilvawlook</code>

Note: Windows XP look is not mentioned in the above table, because this look is available only for platforms running the Microsoft Windows XP operating system. For more details, see the section *Using the Windows XP Look and Feel* on page 222.

For example, if you are building a program using the IBM ILOG Views standard gadget library (See the section *Gadgets Libraries* on page 198 for details) on Microsoft Windows 95, you will need to link with `ilvw95look.lib`, `ilvwlook.lib`.

Similarly, if you are building a program using the IBM ILOG Views advanced gadget library on UNIX, you will need to link with `libilvmlook` and `libilvamlook`.

However, if you are using shared libraries, you can avoid linking with look-and-feel libraries. See the section *Dynamic Loading of Look and Feel* on page 220.

Note: On Windows platforms, linking with the look-and-feel libraries is not required, even when using static libraries. The libraries needed by the application will be automatically linked with it thanks to specific directives put in header files.

Using Several Look and Feel

If you want to use several styles in your program, you must add a compiler option or an `include` file to indicate which of the additional styles you want to use:

- ◆ In the compiler flags, define the symbol names for the styles you want to use:
 - `ILVMOTIFLOOK` for the Motif® look in Windows® applications.

- `ILVWINDOWSLOOK` for the Microsoft® Windows® 3.11 look in X Window applications.
- `ILVWINDOWS95LOOK` for the Microsoft Windows 95 look in X Window applications.
- `ILVWINDOWSXPLOOK` for the Microsoft Windows XP look in Windows applications.
- ◆ In your implementation file, and before any other `#include` directive, include the style header file. Note that if a header file declaring a gadget class precedes one of these header files, the corresponding virtual styles will not be loaded in your program. This may result in a crash when you change the style of your application.

Here are the files to include:

- `<ilviews/motif.h>`
to add an access to the Motif look for Microsoft Windows applications.
- `<ilviews/windows.h>`
to add an access to the Microsoft Windows 3.11 look for X Window applications.
- `<ilviews/win95.h>`
to add an access to the Microsoft Windows 95 look for X Window applications.
- `<ilviews/winxp.h>`
to add an access to the Microsoft Windows XP look. This look is only available for Windows XP platforms.

You must also link your application with the look-and-feel gadget libraries corresponding to the looks used by your application. See Table 9.1 on page 218 and Table 9.2 on page 219.

However, if you are using shared libraries, you can avoid linking with look-and-feel libraries. See the section *Dynamic Loading of Look and Feel* on page 220.

Note: *If you do not want the default look and feel to be used, you must compile with the `ILVNODEFAULTLOOK` flag. Compiling with this flag will prevent you from linking with the default look-and-feel libraries.*

Dynamic Loading of Look and Feel

When using the dynamic loading of look and feel, you do not have to care about which look and feel your application will use. Depending on what is needed by the application, the looks will be loaded at runtime. This means that you do not have to link your application with any look-and-feel specific library.

How does it work?

- ◆ You must use shared libraries (or DLL for Microsoft® Windows®). This is the sine qua none condition without which dynamic loading of modules is not possible.

- ◆ You must not include look specific header files, or you will have to link with the corresponding libraries.
- ◆ You must compile with the `ILVNODEFAULTLOOK` symbol defined, or you will have to link with the default look libraries.

Note: *Using the dynamic loading of look and feel is strongly encouraged, as it allows the application to be completely independent of the style used at runtime.*

Changing the Look and Feel Dynamically

The appearance of a graphic object is managed at different levels:

- ◆ Object level

The method `IlvGraphic::getLookAndFeelHandler()` is used to query an object about its look-and-feel handler. The default implementation is to use the look-and-feel handler defined by the object holder.

- ◆ Holder level

The method `IlvGraphicHolder::getLookAndFeelHandler()` is used to query a holder about its look-and-feel handler. The default implementation is to use the look-and-feel handler defined by the holder display instance.

- ◆ Display level

The method `IlvDisplay::getLookAndFeelHandler()` is used to query a display instance about its look-and-feel handler. The default value is defined by the platform on which the application has been built. See the section *Using the Default Look and Feel* on page 218 for details.

It is possible to change the look of a single gadget, of a whole container, or of the whole application by using respectively the methods `IlvGadget::setLookAndFeelHandler`, `IlvGadgetContainer::setLookAndFeelHandler`, and `IlvDisplay::setLookAndFeelHandler`.

A look-and-feel handler is a subclass of the `IlvLookAndFeelHandler` class. Each handler has a unique name that identifies it. Here are the names for the four predefined look-and-feel styles:

Motif	motif
Windows 3.11	windows
Windows 95	win95
Windows XP	winxp

A look-and-feel handler is associated to a display instance. To obtain an instance of a look-and-feel handler, use the method `IlvDisplay::getLookAndFeelHandler` that takes a name as parameter:

```
IlvLookAndFeelHandler* lfh = display->getLookAndFeelHandler(IlGetSymbol("motif"));
```

If the required look and feel has already been created for this display, it is returned, otherwise a new one is created. If the look and feel cannot be created, the method returns 0.

Changing the Look and Feel of the Whole Display

As seen above, the method `IlvDisplay::setLookAndFeelHandler` should be used to change the look and feel of the whole display. However, the ILOG Views 4.0 API used to define an enum (`IlvLookAndFeelStyle`) to describe predefined looks. This enum can still be used as follow:

```
#include <ilviews/ilv.h>

typedef enum IlvLookAndFeelStyle {
    IlvOtherLook,
    IlvMotifLook,
    IlvWindowsLook,
    IlvWindows95Look,
    IlvWindowsXPLook
};
```

The following member functions of the `IlvDisplay` class let you manipulate the setting of the look display resource using this enum:

- ◆ `IlvDisplay::getCurrentLook` returns the current style identifier used by this display instance. If the current look and feel of the display is not one of the predefined look-and-feel styles, `IlvOtherLook` is returned.
- ◆ `IlvDisplay::setCurrentLook` sets the style identifier used by this display instance to style.

You can be informed of a change in the look and feel of the display by using the following methods:

- ◆ `IlvDisplay::addChangeLookCallback` and lets you add user-defined functions that are called when the style is dynamically changed.
- ◆ `IlvDisplay::removeChangeLookCallback` lets you remove user-defined functions that are called when the style is dynamically changed.

Using the Windows XP Look and Feel

Although the Microsoft Windows 3.11, Microsoft Windows 95, and Motif styles are independent of the platform, the Microsoft Windows XP style uses the system (Microsoft Windows XP) to draw the components. This means that you can only use this style on a

platform running Microsoft Windows XP. This also means that if a new theme is available to Microsoft Windows XP, it will also be available to IBM ILOG Views applications.

Note: You can build an application using the Microsoft Windows XP style on any other Microsoft Windows platform. In this case, you may need to install the Microsoft Platform SDK. To get the SDK, go to <http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en>.

Using IBM ILOG Views Dynamic Link Libraries

When using the IBM ILOG Views DLLs (`dll_md`, or `dll_mda`), the Microsoft Windows XP style can be dynamically loaded when needed. This means that you can build an IBM ILOG Views application that can be run on any Microsoft Windows platform. The application will load the Microsoft Windows XP style if it is needed. See *Dynamic Loading of Look and Feel* on page 220 for more details.

Using IBM ILOG Views Static Libraries

When using the IBM ILOG Views static libraries (`stat_st`, `stat_sta`, `stat_md`, `stat_mda`, `stat_mt`, or `stat_mta`), you must be aware of the following issues:

- ◆ When building an IBM ILOG Views application using the default style on a Microsoft Windows XP platform, the flag `WINVER` must be set to `0x501` when compiling your application, otherwise only the Microsoft Windows 95 style will be registered. See *Using the Default Look and Feel* on page 218.
- ◆ An IBM ILOG Views application using the Microsoft Windows XP style and the IBM ILOG Views static libraries can be run only on platforms running Microsoft Windows XP. If you want to compile an IBM ILOG Views application on a Microsoft Windows XP platform and you want this application to run on any Windows platform, you can either not define the `WINVER` flag, or define the `ILVNODEFAULTLOOK` flag. In this last case, you will need to link your application with look-and-feel libraries other than XP libraries. See *Using Several Look and Feel* on page 219 for more details.
- ◆ As the dynamic loading of modules is disabled when using static libraries, your application needs to be linked with the right libraries. See *Look Libraries for Windows Platforms* on page 218.

Dialogs

The Gadgets library provides the `IlvDialog` class that you can use to create dialog boxes. Since this class inherits from `IlvGadgetContainer`, its instances can contain gadgets. `IlvDialog` has various subclasses that implement standard dialog boxes.

This chapter covers the following topics:

- ◆ *Predefined Dialog Boxes*
- ◆ *Creating Your Own Dialog Box*
- ◆ *Showing and Hiding Dialog Boxes*
- ◆ *Setting a Default Button*

The following illustration shows the dialog class hierarchy:

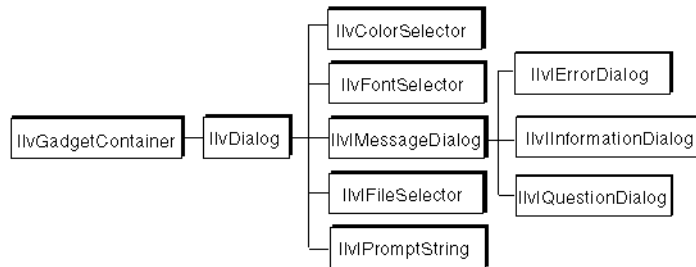


Figure 10.1 Class Hierarchy of Dialog Boxes

Predefined Dialog Boxes

The Gadgets library provides the following classes for defining standard dialog boxes:

- ◆ *IlvMessageDialog*
- ◆ *IlvQuestionDialog*
- ◆ *IlvErrorDialog*
- ◆ *IlvWarner*
- ◆ *IlvInformationDialog*
- ◆ *IlvFileSelector*
- ◆ *IlvPromptString*
- ◆ *IlvFontSelector*
- ◆ *IlvColorSelector*

IlvMessageDialog

A message dialog box (*IlvMessageDialog* class) includes a message text field, a bitmap, and two buttons.

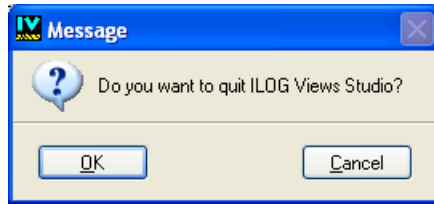


Figure 10.2 A Message Dialog Box

Note: By default, this dialog does not include a bitmap. Therefore, you have to provide one.

IlvQuestionDialog

A question dialog box (`IlvQuestionDialog` class) displays a question and expects a yes or no answer.

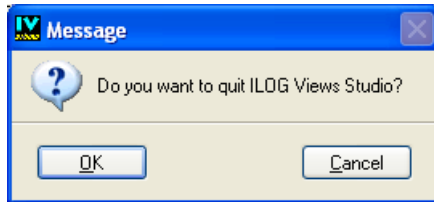


Figure 10.3 A Question Dialog Box

Here is a code example for a question dialog box:

```
{
    IlvQuestionDialog dlg(getDisplay(), msg, 0,
                          IlvDialogOkCancel, transientFor);
    dlg.setString("dialog message");
    if (dlg.get()) ...
}
```

This code creates a dialog box named `dlg` that will be destroyed after its use. This dialog box will be transient for the view specified by `transientFor`. It has two buttons, Ok and Cancel. The method `get` opens the dialog box and waits for the result. This method returns `IlvTrue` if Ok is chosen, and `IlvFalse` otherwise.

IlvErrorDialog

An error dialog box (`IlvErrorDialog` class) displays an error message.



Figure 10.4 An Error Dialog Box

IlvIWarner

A warning dialog box (`IlvIWarner` class) displays a warning message.



Figure 10.5 A Warning Dialog Box

IlvIInformationDialog

An information dialog box (`IlvIInformationDialog` class) displays an information message.

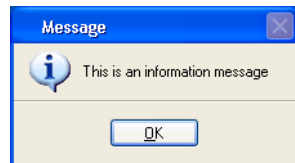


Figure 10.6 An Information Dialog Box

IlvIFileSelector

A file selector (`IlvIFileSelector` class) asks the user to select a file name.

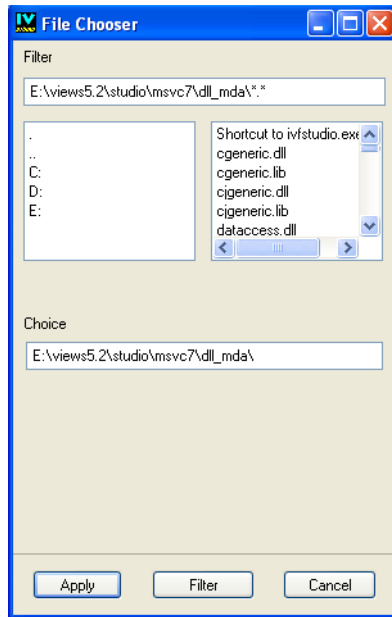


Figure 10.7 A File Selector

Here is an example of using a file selector:

```
filesel = new IlvIFileSelector(display, 0, "*.cc");
filesel->setName("File Chooser");
filename = filesel->get();
if (filename && filename[0] && IlvFileExists(filename)) ...
```

Note: If you want to use the file selector specific to the platform you are working on, use either the `IlvFileSelector` or the `IlvFileBrowser` class.

IlvIPromptString

A prompt string (`IlvIPromptString` class) asks the user to select or to type a string.

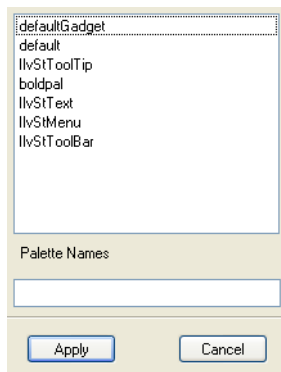


Figure 10.8 A Prompt String

IlvFontSelector

A font selector (`IlvFontSelector` class) asks the user to select a font.

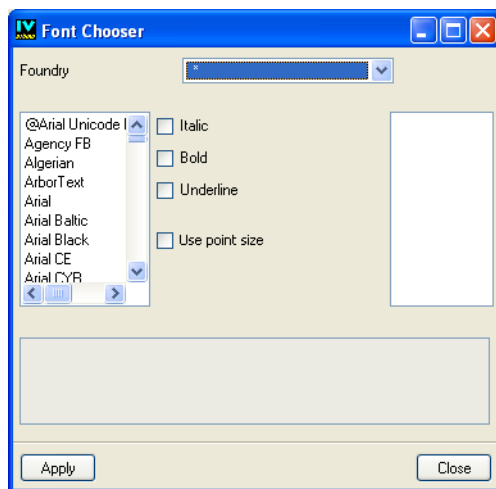


Figure 10.9 A Font Selector

IlvColorSelector

A color selector (`IlvColorSelector` class) asks the user to select a color.

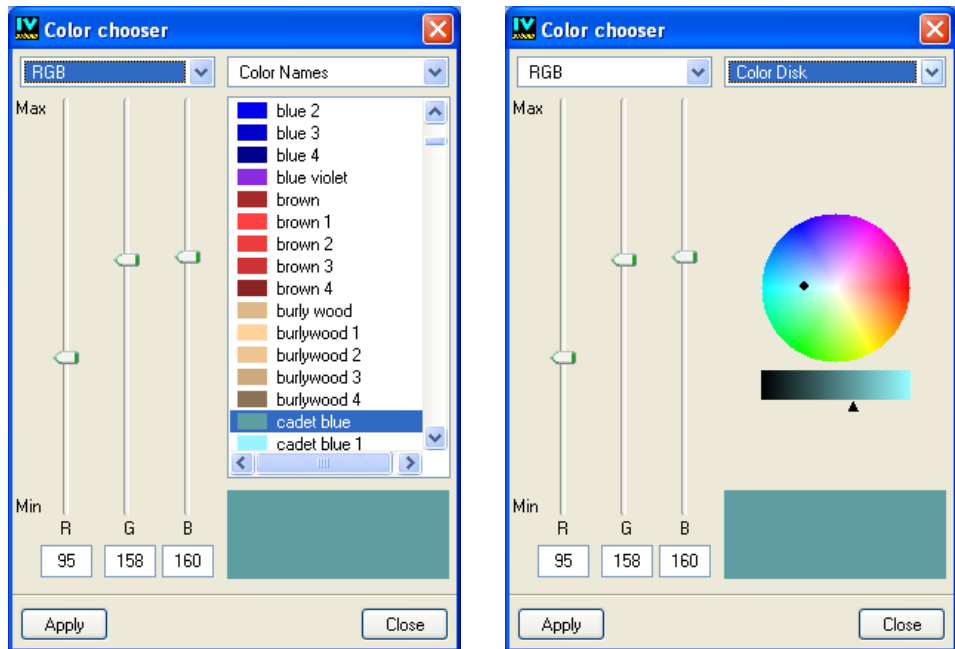


Figure 10.10 Color Selectors with Color Names (Left) and with a Color Wheel (Right)

Creating Your Own Dialog Box

To create your own dialog box, follow the steps below:

1. Design the visual representation of the panel.

This step includes several aspects such as choosing which gadgets to use, how the focus is managed, or how gadgets will behave when the dialog is resized. This phase can be achieved by using IBM® ILOG® Views Studio.

2. Display your panel in a dialog box.

You can either use the generated code of IBM ILOG Views Studio (For details, see Chapter 4, *Using the Generated Code*), or directly use the `IlvDialog` class. The constructors of `IlvDialog` provide a facility to pass a file name as a parameter.

The class `IlvDialog` has already two registered callbacks:

- ◆ `apply`: This callback invokes the virtual `IlvDialog::apply` method.
- ◆ `cancel`: This callback invokes the virtual `IlvDialog::cancel` method.

You can, for example, have two buttons in your dialog box, one with the callback set to “apply” and the other set to “cancel”.

Showing and Hiding Dialog Boxes

The class `IlvDialog` provides methods for managing dialog boxes.

Use the `IlvDialog` method `wait` to wait until the user clicks Ok or Cancel (which calls the Apply or Cancel callbacks). This method displays a modal dialog box. The method `wasCanceled` tells you whether the user has clicked Cancel.

```
dialog.wait();
if (!dialog.wasCanceled()) {
    ...
}
```

You can also use the methods `show` and `hide`. Standard dialog boxes have their own special methods that display them and wait until the value is returned.

Setting a Default Button

A dialog box can have a default button. The default button is the one that is activated when the user presses the Enter key when the dialog box has the keyboard focus.

The default button has a special appearance that distinguishes it from other buttons. To set a default button, use the `setDefaultButton` member function.

Note that when a default button has been defined, pressing the Enter key only applies to this button. In certain cases you might want to override this behavior. For example, when editing a matrix you might want to use the Enter key to validate changes. To modify this behavior, you can use the member function `IlvGadget::usesDefaultButtonKeys`.

Using Common Gadgets

This chapter explains how to use the large variety of gadgets provided in the Gadgets library. It covers the following topics:

- ◆ *Using IlvArrowButton*
- ◆ *Using IlvButton*
- ◆ *Using IlvComboBox and IlvScrolledComboBox*
- ◆ *Using IlvDateField*
- ◆ *Using IlvFrame*
- ◆ *Using IlvMessageLabel*
- ◆ *Using IlvNotebook*
- ◆ *Using IlvNumberField*
- ◆ *Using IlvOptionsMenu*
- ◆ *Using IlvPasswordTextField*
- ◆ *Using IlvScrollBar*
- ◆ *Using IlvSlider*
- ◆ *Using IlvSpinBox*
- ◆ *Using IlvStringList*

- ◆ *Using IlvText*
- ◆ *Using IlvTextField*
- ◆ *Using IlvToggle*
- ◆ *Using IlvTreeGadget*

Using IlvArrowButton

The class `IlvArrowButton` defines a button displaying an arrow. The arrow can be oriented up, down, right, or left. `IlvArrowButton` is a subclass of `IlvButton`.



Figure 11.1 Arrow Buttons

You can specify the direction of an arrow using `IlvArrowButton::setDirection` and retrieve it with `IlvArrowButton::getDirection`.

See *Using IlvButton* on page 233.

Using IlvButton

The class `IlvButton` defines a rectangular area that the user can click. `IlvButton` is a subclass of `IlvMessageLabel` around which it adds a relief rectangle.

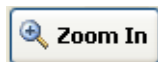


Figure 11.2 A Button

The label that appears inside a button can have various alignment settings and can be localized. For details about these properties, see *Using IlvMessageLabel* on page 239.

This section covers the following topics:

- ◆ *Displaying a Bitmap in a Button*
- ◆ *Displaying the Button Frame*
- ◆ *Associating a Mnemonic with a Button*
- ◆ *Event Handling and Callbacks*

Displaying a Bitmap in a Button

A button can display a bitmap. Four kinds of bitmaps can be displayed: sensitive, nonsensitive, selected, and highlighted.

Sensitive bitmaps are described in *Associating a Bitmap with a Message Label* on page 240.

A selected bitmap is displayed when the button is clicked. To set a selected bitmap, use `IlvButton::setSelectedBitmap`.

A highlighted bitmap is displayed when the mouse is over the button. To set a highlighted bitmap, use `IlvButton::setHighlightedBitmap`.

Displaying the Button Frame

You can use the `IlvButton::showFrame` member function to specify whether or not the frame surrounding the button be displayed when it is highlighted. The following illustration shows buttons in the Windows® 95 look and feel.



Figure 11.3 Button with Frame Hidden (Left) and with Frame Displayed (Right)

Associating a Mnemonic with a Button

A button label can be associated with a mnemonic letter. When you press the key corresponding to the mnemonic letter, the `IlvButton::activate` member function is called. If the button does not have the keyboard focus, you must press the modifier key (Alt on PCs and Meta on UNIX) with the letter.

See *Associating a Mnemonic with a Gadget Label* on page 212.

Event Handling and Callbacks

When the user clicks a button or presses the mnemonic letter associated with it, or presses the Enter key or the space bar, the `IlvButton::activate` member function is called. This virtual member function calls the Main callback of the button.

See *Handling Events* on page 204 and *Associating a Callback with a Gadget* on page 210.

Using IlvComboBox and IlvScrolledComboBox

The class `IlvComboBox` combines a text field with a list of predefined strings from which the user can choose. `IlvScrolledComboBox` displays a scrollbar when the list exceeds a certain number of choices. `IlvComboBox` class is a subclass of `IlvTextField` and `IlvListGadgetItemHolder`, and `IlvScrolledComboBox` class is a subclass of `IlvComboBox`.



Figure 11.4 A Combo Box

See *Using IlvTextField* on page 264.

This section covers the following topics:

- ◆ *Setting a Combo Box as Noneditable*
- ◆ *Setting and Retrieving Items*
- ◆ *Changing or Retrieving the Selection*
- ◆ *Using Large Lists*
- ◆ *Setting the Number of Visible Items*
- ◆ *Localizing Combo Boxes*
- ◆ *Event Handling and Callbacks*

Setting a Combo Box as Noneditable

By default, the text field part of the combo box can be edited, which means that you can change its content either by typing new text in it or pasting text from the clipboard. The member function `IlvTextField::setEditable` allows you to switch to read-only mode. In this mode, you can only choose a value from the menu.

The appearance of a combo box changes when it switches to read-only mode. Therefore, you must call the `IlvGadget::redraw` member function when changing the editing mode of a combo box.

Setting and Retrieving Items

Because `IlvComboBox` is a subclass of `IlvListGadgetItemHolder`, you must use the member functions of this class to modify the items of the combo box.

See `IlvListGadgetItemHolder`.

Changing or Retrieving the Selection

Because `IlvComboBox` is a subclass of `IlvTextField`, you can use the member functions of this class to set the text in a combo box or retrieve it.

See *Setting and Retrieving Text* on page 265.

You can also set the selected item in a combo box using an index number with `IlvComboBox::setSelected` and retrieve it with `IlvComboBox::whichSelected`.

Using Large Lists

Unlike `IlvComboBox`, the item list displayed by `IlvScrolledComboBox` has a fixed width corresponding to the combo box width. If the list contains large items, they might not fit in the text field and thus in the list. To modify this behavior, use the member function `IlvScrolledComboBox::enableLargeList`.

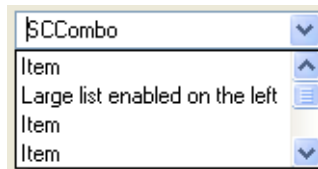


Figure 11.5 A Combo Box with Large List Enabled

Setting the Number of Visible Items

In a scrolled combo box, you can fix the number of visible items in the list. If all items are visible, there is no scrollbar.

To set the number of visible items, use the member function `IlvScrolledComboBox::setVisibleItems`.

Localizing Combo Boxes

The text appearing in a combo box can be localized. Only noneditable combo boxes can be localized.

See *Localizing a Gadget* on page 211.

Event Handling and Callbacks

When the user selects a new item with the mouse, uses the arrow keys, or enters new text and presses the Enter key, the Main callback of the combo box is invoked.

When the user opens the combo box list, the Open List callback is invoked. To set an Open List callback method, use the callback symbol returned by `IlvComboBox::OpenListSymbol()`.

See *Associating a Callback with a Gadget* on page 210.

Using IlvDateField

The class `IlvDateField` defines a special text field gadget for editing dates with various formats. `IlvDateField` is a subclass of `IlvTextField`.




Figure 11.6 A Date Field

This section covers the following topics:

- ◆ *Formatting a Date*
- ◆ *Setting and Retrieving a Date Value*

Formatting a Date

The `IlvDateField` can display dates in many formats. To specify the date format, use the member function `IlvDateField::setFormat`.

A date is composed of three elements: the day, the month, and the year. These elements are divided by separation characters. The `setFormat` member function allows you to specify these elements and which separator to use.

The default value is: 12/31/1995 (`df_Month`, `df_Day`, `df_Year`).

The formats are defined as follows:

```
enum format
{
    df_day,           // 1
    df_Day,          // 01

    df_month,        // 3
    df_Month,        // 03
    df_month_text,   // March
    df_abbrev_month, // Mar

    df_year,         // 95
    df_Year          // 1995
};
```

- ◆ `df_day` Writes the day as a number with no leading zero.

- ◆ `df_Day` Writes the day as a number with a leading zero, if necessary.
- ◆ `df_month` Writes the month as a number with no leading zero.
- ◆ `df_Month` Writes the month as a number with a leading zero, if necessary.
- ◆ `df_month_text` Writes the month name. If the following month names appear in the language database, the corresponding name is taken from it. Otherwise, the month name is taken with the character ‘&’ removed.

Month names: &January, &February, &March, &April, &May, &June, &July, &August, &September, &October, &November, &December.

- ◆ `df_abbrev_month` Writes the abbreviated month name. If the following abbreviated month names appear in the language database, the corresponding name is taken from it. Otherwise, the abbreviated month name is taken with the character ‘&’ removed.

Abbreviated month names: &january, &february, &march, &april, &may, &june, &july, &august, &september, &october, &november, &december.

- ◆ `df_year` Writes the last two digits of the year.
- ◆ `df_Year` Writes the year.

If you change the format when the field contains a value, this value is applied the new format.

***Note:** Only a single day, year, or month format can be passed to the `setFormat` member function. Otherwise, the function returns `IlvFalse` and the format remains unchanged.*

Formats are defined in an embedded enum declaration. They are set as follows:

```
obj->setFormat (IlvDateField::df_day) ;
```

Examples of Formats

```
April,2,1995 (df_month_text, df_day, df_Year with separator ,)
```

```
2/4/95 (df_day, df_month, df_year with separator /)
```

```
02/04/1995 (df_Day, df_Month, df_Year with separator /)
```

Setting and Retrieving a Date Value

To set the date of an `IlvDateField` or retrieve it, use the member functions `IlvDateField::setValue` and `IlvDateField::getValue`.

Year 2000 Management

The right way to avoid problems linked to the new millenium is to use four digits to represent the year. This can be done in the `IlvDateField` class by using the `setFormat` member function.

However, if you must use a two-digits value to represent the year part of the date, the `IlvDateField` API offers several methods to solve the problem:

1. `SetBaseCentury` lets you specify the base century that will be used to recompute the full year
2. `GetBaseCentury` returns the base century set by `SetBaseCentury`. The default value is 1900
3. `SetCenturyThreshold` lets you specify the threshold over which the base century used will be the value returned by `GetBaseCentury()` plus 1
4. `GetCenturyThreshold` returns the value set by `SetCenturyThreshold`. The default value is 30

For example, with a base century of 1900 and a threshold of 30, a value of 10 is converted to 2010, and a value of 40 is converted to 1940.

Using `IlvFrame`

The class `IlvFrame` displays a rectangle around a label. It is used for grouping gadgets together in a section of a panel. `IlvFrame` derives from the class `IlvMessageLabel`.



Figure 11.7 A Frame

See *Using `IlvMessageLabel`* on page 239.

Associating a Mnemonic with a Frame

Frame labels can be associated with a mnemonic letter. When you press the key corresponding to the mnemonic letter with the modifier key (Alt on PCs and Meta on UNIX), the keyboard focus is given to the first gadget in the frame that can have the focus.

See *Associating a Mnemonic with a Gadget Label* on page 212 and *Focus Management* on page 204.

Using `IlvMessageLabel`

The class `IlvMessageLabel` displays a message, which can be accompanied by a bitmap. Messages are recorded in a database that can be associated with the current instance of `IlvDisplay`.

The alignment of the message relative to the bitmap can be set to any position. In addition, it is possible to change the alignment of the whole block (message + bitmap) to `IlvCenter`, `IlvLeft`, or `IlvRight` in relation to its bounding box.

Figure 11.8 shows an example of an `IlvMessageLabel`. The alignment of the message relative to the picture is `IlvBottom`, and the global alignment of the `IlvMessageLabel` is `IlvCenter`.

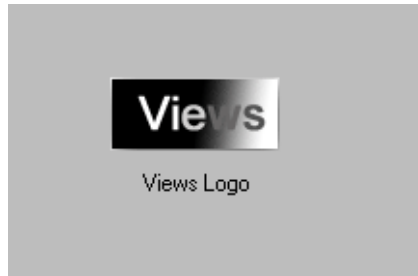


Figure 11.8 A Message Label

This section covers the following topics:

- ◆ *Associating a Bitmap with a Message Label*
- ◆ *Making the Message Label Opaque*
- ◆ *Laying Out the Message Label*
- ◆ *Localizing a Message Label*
- ◆ *Associating a Mnemonic*

Associating a Bitmap with a Message Label

Bitmaps can be associated with a message label using the member functions `IlvMessageLabel::setBitmap` and `IlvMessageLabel::setInsensitiveBitmap`.

`setBitmap` associates a main bitmap with the message label. `setInsensitiveBitmap` sets the bitmap that will be displayed when the message label is set to nonsensitive. If you do not provide a nonsensitive bitmap, a default one is automatically computed from the sensitive bitmap when setting the message label to nonsensitive.

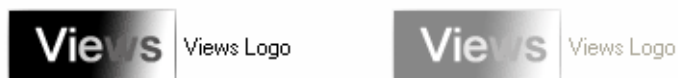


Figure 11.9 Message Label with Sensitive and Nonsensitive Bitmap

See *Localizing a Gadget* on page 211.

Making the Message Label Opaque

Unlike the other gadgets, the message label is transparent by default. To make it opaque, call the member function `IlvGadget::setTransparent` with `ILFalse` as its parameter. The bounding box of an opaque message label is filled with the background color that is set in the object palette.



Figure 11.10 *An Opaque Message Label*

See *Setting a Gadget as Transparent* on page 209.

Laying Out the Message Label

When a message label displays both a label and a bitmap, you can change the position of the label relative to the bitmap using the member function `IlvMessageLabel::setLabelPosition`.

For example, to have the label appear to the left of the bitmap, call:

```
message->setLabelPosition(ILvLeft);
```

To set the spacing between the label and the bitmap to 20 pixels, call:

```
message->setSpacing(20);
```

To center the grouped label and the bitmap inside the bounding box of the message label, call:

```
message->setAlignment(ILvCenter);
```



Figure 11.11 *Label and Bitmap Aligned as One Block*

Localizing a Message Label

A message label can be localized.

See *Localizing a Gadget* on page 211.

Associating a Mnemonic

Message labels can include a mnemonic letter. When you press the key corresponding to the mnemonic letter with the modifier key (Alt on PCs or Meta on UNIX), or click the message label, the focus is given to the next gadget in the focus chain.

See *Associating a Mnemonic with a Gadget Label* on page 212.

Using IlvNotebook

The class `IlvNotebook` simulates a real notebook. A notebook is composed of pages that you can select and bring to the front by clicking their tab. These pages are implemented by the class `IlvNotebookPage`.

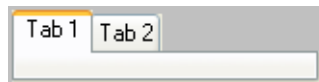


Figure 11.12 A Notebook with Pages

This section covers the following topics:

- ◆ *Customizing Notebook Tabs*
- ◆ *Handling Notebook Pages*
- ◆ *Event Handling and Callbacks*

Customizing Notebook Tabs

The tabs of a notebook can be customized in many different ways:

- ◆ *Setting the Position of Tabs*
- ◆ *Setting the Orientation of Tabs*
- ◆ *Setting the Tabs Margins*
- ◆ *Setting the Page Margins*

Setting the Position of Tabs

The tabs of a notebook can be displayed on any of its borders (top, bottom, left, or right). You can change the position of the tabs with the member function `IlvNotebook::setTabsPosition` and retrieve this position with `IlvNotebook::getTabsPosition`.

Setting the Orientation of Tabs

Within the tab, the label can be drawn horizontally or vertically. You can change the orientation of the labels with the member function `IlvNotebook::setLabelsVertical`. To know whether the labels are horizontal or vertical, use `IlvNotebook::areLabelsVertical`.

When the tab labels are oriented vertically, the label can be written from top to bottom, or from bottom to top.

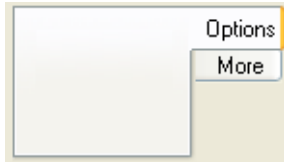


Figure 11.13 A Notebook with Vertical Tabs

A label that is written from bottom to top is said to be flipped. To change the way vertical labels are drawn, use these member functions `IlvNotebook::mustFlipLabels` and `IlvNotebook::flipLabels`.

Setting the Tabs Margins

You can change the margin between the border of the tab and its label with these member functions: `IlvNotebook::getXMargin`, `IlvNotebook::setXMargin`, `IlvNotebook::getYMargin`, and `IlvNotebook::setYMargin`.

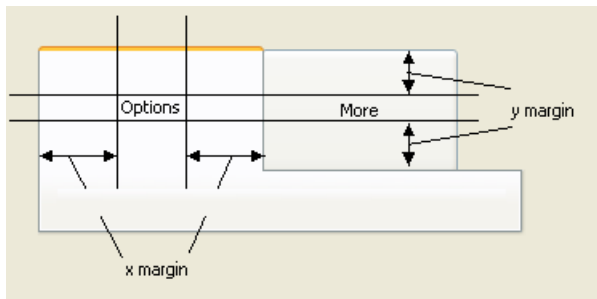


Figure 11.13 Tab Margins (x and y)

Setting the Page Margins

You can also change the margins between the border of the notebook and its page view. The value returned by the member function `IlvNotebook::getPageArea` depends on the values set for the page margins.

Following are the member functions for setting the page margins:

`IlvNotebook::setPageTopMargin`, `IlvNotebook::setPageBottomMargin`, `IlvNotebook::setPageLeftMargin`, `IlvNotebook::setPageRightMargin`. You can retrieve the margins set with the corresponding get member functions.

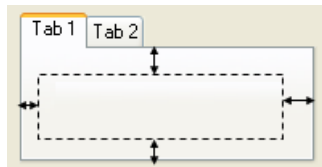


Figure 11.14 Page Margins

Handling Notebook Pages

The pages of a notebook are implemented by the class `IlvNotebookPage`, which you can subclass to meet specific requirements. Instances of `IlvNotebookPage` can encapsulate an `IlvGadgetContainer` or any other type of view. See *Displaying the Contents of a Page* on page 245.

This section covers the following topics:

- ◆ *Adding and Removing a Notebook Page*
- ◆ *Displaying the Contents of a Page*
- ◆ *Customizing a Notebook Page*
- ◆ *Changing the Color of a Notebook Page*
- ◆ *Setting the Content of Tabs*

Adding and Removing a Notebook Page

When created, a notebook has no pages. A notebook must contain at least one page.

To add a page to a notebook, use one of the `IlvNotebook::addPage` member functions:

```
IlvNotebookPage* addPage(IlvNotebookPage* page,
                        IUShort idx = IlvNotebookLastPage);

IlvNotebookPage* addPage(const char* label,
                        IlvBitmap* bitmap = 0,
                        IlBoolean transparent = IlTrue,
                        const char* filename = 0,
                        IUShort idx = IlvNotebookLastPage);
```

The first `addPage` member function lets you add a subclass of `IlvNotebookPage`. The second one creates a new instance of `IlvNotebookPage`. The `idx` parameter specifies the position at which the page is to be added.

An `IlvNotebookPage` is always related to a specific notebook, which means that you cannot share an `IlvNotebookPage` between two notebooks. You can retrieve the notebook related to a page using `IlvNotebookPage::getNotebook`.

To know how many pages there are in a notebook, use `IlvNotebook::getPagesCardinal`.

To retrieve the internal notebook page array, use `IlvNotebook::getPages`. To retrieve the first page, call:

```
page = notebook->getPages()[0];
```

To remove a specific page, use `IlvNotebook::removePage`.

Displaying the Contents of a Page

The member function `IlvNotebookPage::createView` creates a view of type `IlvGadgetContainer` to display the contents of the page, which you can retrieve with `IlvNotebookPage::getView`.

You can load an `.ilv` file into a notebook page with the member function `IlvNotebookPage::setFileName` and retrieve this file with `IlvNotebookPage::getFileName`.

The member function `setFileName` assumes that the view is an `IlvGadgetContainer` or one of its subclasses. You will have to override it if you use another type of view.

Customizing a Notebook Page

You can change the view held by an `IlvNotebookPage` using the member function `IlvNotebookPage::setView`. You can also redefine the member function `IlvNotebookPage::createView` in a subclass of `IlvNotebookPage`. See *Displaying the Contents of a Page* on page 245.

This member function instantiates an invisible view with the size given as its parameter. It also loads an `.ilv` file (result of `getFileName`) into the new view.

For example, to have notebook pages encapsulate `IlvScrolledView` instances, subclass `IlvNotebookPage` as follows:

```
class myNotebookPage : public IlvNotebookPage
{
public:
    myNotebookPage(IlvNotebook* gadget,
                  const char*   label,
                  Ilvbitmap*    bitmap,
                  IlBoolean     transparent,
                  const char*   filename)
        : IlvNotebookPage(gadget, label, bitmap, transparent, filename) {}
};
```

```

        virtual IlvView* createView(IlvAbstractView* parent,
            const IlvRect& size);
};

IlvView* myNotebookPage::createView(IlvAbstractView* parent,
            const IlvRect& size)
{
    IlvScrolledView* scview = new IlvScrolledView(parent,size);
    IlvGadgetContainer* child = new IlvGadgetContainer(scview->getClipView(),
            IlvRect(0,0,100,100));

    if (_filename && _filename[0])
        child->readFile(_filename);
    child->fitToContents();
    return scview;
}

```

Then create a new notebook page and add it to the notebook:

```

myNotebookPage* np5=
    new myNotebookPage(nb, "Page5", 0, IlvFalse, "../snbook.ilv");
nb->addPage(np5);

```

If your view can read an .ilv file, you can overload the member function `IlvNotebookPage::setFileName`.

If you need to add more drawings to your page, you can overload the `draw` method:

```

void draw(IlvPort* dst,
        const IlvRect& pageRect,
        const IlvTransformer* t,
        const IlvRegion* clip) const;

```

You also need to create the following constructors for your page:

```

MyNotebookPage::MyNotebookPage(IlvNotebook* notebook);
MyNotebookPage::MyNotebookPage(IlvNotebook* notebook,
                                const char* label,
                                IlvBitmap* bitmap,
                                IlvBoolean transparent,
                                const char* filename);
MyNotebookPage::MyNotebookPage(const MyNotebookPage& source);
MyNotebookPage::MyNotebookPage(IlvNotebook* notebook,
                                IlvInputFile&);

```

The member function `IlvNotebookPage::write` and the constructor that takes an `IlvInputFile` as a parameter let you extend the .ilv format of the page.

Changing the Color of a Notebook Page

Each page of the notebook can have a different background color. To change this color, use the member function `IlvNotebookPage::setBackground`. When you change the background color of a notebook page, this color is applied to the background of its view.

Setting the Content of Tabs

The notebook tab can contain a label and/or a bitmap. To set the label that appears in a notebook tab, use the member function `IlvNotebookPage::setLabel`. Use `IlvNotebookPage::getLabel` to retrieve this label.

This label can have a mnemonic. See *Associating a Mnemonic with a Gadget Label* on page 212.

To set the bitmap that appears in a notebook tab, use the member function `IlvNotebookPage::setBitmap`. Use `IlvNotebookPage::getBitmap` to retrieve this bitmap.

Event Handling and Callbacks

When the user selects a notebook page by clicking it with the mouse, by pressing the arrow keys, or by pressing the key corresponding to its associated mnemonic letter, the member function `IlvNotebook::changeSelection` is called. This member function invokes `IlvNotebook::pageDeselected` with the previous selected page as its parameter and `IlvNotebook::pageSelected` with the new selected page as its parameter. `IlvNotebook::pageDeselected` calls `IlvNotebookPage::deselect` and triggers the Page Deselected callback. `IlvNotebook::pageSelected` calls `IlvNotebookPage::select` and triggers the Page Selected callback. You can retrieve their types with `IlvNotebook::PageSelectedCallbackType` and `IlvNotebook::PageDeselectedCallbackType`. Resizing the notebook page invokes the Page Resize callback. You can retrieve its type with `IlvNotebook::PageResizedCallbackType`.

See *Associating a Callback with a Gadget* on page 210.

Using IlvNumberField

The class `IlvNumberField` class defines a specialized text field for editing numbers with various formats. `IlvNumberField` is a subclass of `IlvTextField`.

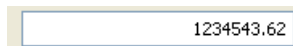


Figure 11.15 A Number Field

See *Using IlvTextField* on page 264.

This section covers the following topics:

- ◆ *Selecting an Editing Mode*
- ◆ *Choosing a Format*

- ◆ *Defining a Range of Values*
- ◆ *Setting and Retrieving a Value*
- ◆ *Specifying the Thousand Separator*
- ◆ *Specifying the Decimal Point Character*
- ◆ *Event Handling and Callbacks*

Selecting an Editing Mode

The `IlvNumberField` class has two main editing modes, one for integers (`IlvInt`) and one for floating-point numbers (`IlDouble`). The editing mode in effect depends on the constructor used.

To create a number field for editing integers, use one of these two constructors:

```
IlvNumberField* field = new IlvNumberField(display, 0,
                                           IlvRect(10,10, 100, 30));
IlvNumberField* field = new IlvNumberField(display,
                                           IlvPoint(10,10), 0);
```

To create a number field for editing floating point numbers, use one of these two constructors:

```
IlvNumberField* field = new IlvNumberField(display, 0.0,
                                           IlvRect(10,10, 100, 30));
IlvNumberField* field = new IlvNumberField(display,
                                           IlvPoint(10,10), 0.0);
```

Choosing a Format

A number field can be assigned a format. You can change the current format at runtime with `IlvNumberField::setFormat`.

Formats are defined by this enum declaration:

```
enum { thousands = 1,
      scientific = 2,
      padright   = 4,
      showpoint  = 8,
      floatmode  = 16};
```

They are set as follows:

```
obj->setFormat(IlvNumberField::floatmode|IlvNumberField::scientific);
```

Following is a description of these various date formats:

- ◆ `floatmode`—Use this mode to edit float values. This mode is automatically set when using a constructor with a value of type `IlDouble`. See *Selecting an Editing Mode* on page 248.

- ◆ `scientific`—Use this mode to control the output of `IlDouble` values in the gadget. If `scientific` is set, the value is converted using scientific notation, where there is one digit before the decimal point and the number of digits after it is equal to the specified number (six by default). The letter `e` introduces the exponent. If `scientific` is not set, the value is converted to decimal notation with precision digits after the decimal point (six digits by default). This only works when the `floatmode` is set.
- ◆ `padright`—Use this mode to add the trailing zeros after the decimal point. This only works when the `floatmode` is set.
- ◆ `showpoint`—Use this mode with the `padright` mode to keep the decimal point when removing trailing zeros. This only works when the `floatmode` is set.
- ◆ `thousands`—Use this mode to display a “thousand” separator. The default “thousand” separator is the character `’.`. See *Specifying the Thousand Separator* on page 249.

Defining a Range of Values

You can specify the minimum and maximum numbers that can be edited in a number field. There are two sets of member functions depending on whether you are editing an integer or a floating-point value:

For integers, use `IlvNumberField::setMaxInt` and `IlvNumberField::setMinInt`.

For floating-point numbers, use `IlvNumberField::setMaxFloat` and `IlvNumberField::setMinFloat`.

Setting and Retrieving a Value

The `IlvNumberField` provides two sets of member functions for setting and retrieving a value.

If the value is an integer, use:

```
IlInt      getIntValue(IlBoolean& error) const;
IlBoolean setValue(IlInt, IlBoolean redraw = IlFalse);
```

If the value is a floating-point number:

```
IlDouble  getFloatValue(IlBoolean& error) const;
IlBoolean setValue(IlDouble, IlBoolean redraw = IlFalse);
```

Specifying the Thousand Separator

When the number formats `thousands` and `float` are set, the thousand separator is displayed. The default thousand separator is the comma character (`,`). You can change this character using the member function `IlvNumberField::setThousandSeparator`. Calling this member functions does not directly change the text in the number field.

Therefore, if the field already contains a value, you must first retrieve that value, change the separator, and then set the value again.

Specifying the Decimal Point Character

The default decimal point character for floating-point numbers is the period character (.). You can change this character using the member function

```
IlvNumberField::setDecimalPointChar.
```

Calling this member function does not directly change the text in the number field. Therefore, if the field already contains a value, you must first retrieve that value, change the decimal point character, and then set the value again.

Event Handling and Callbacks

When the user presses the Enter Key in a number field, the `IlvNumberField::validate` member function is called. This virtual member function invokes the Main callback associated with the number field and moves the keyboard focus to the next gadget in the focus chain. This happens only if the field content can be converted to a number, and this number is within the range specified by `IlvNumberField::setMaxFloat`, `IlvNumberField::setMinFloat`, `IlvNumberField::setMaxInt`, and `IlvNumberField::setMinInt`.

See *Defining a Range of Values* on page 249, *Focus Management* on page 204, and *Associating a Callback with a Gadget* on page 210.

Using IlvOptionsMenu

The class `IlvOptionsMenu` defines a drop-down list of items from which the user can select.



Figure 11.16 An Option Menu

Note: Because the Microsoft® Windows® look and feel does not provide an option menu, the class `IlvOptionsMenu` is represented as a combo box when the style in use is Microsoft Windows.

This section covers the following topics:

- ◆ *Setting and Retrieving Items*
 - ◆ *Changing and Retrieving the Selected Item*
 - ◆ *Localizing Option Menus*
 - ◆ *Event Handling and Callbacks*
-

Setting and Retrieving Items

Because `IlvOptionMenu` is a subclass of `IlvListGadgetItemHolder`, you must use the member functions of this class to modify the items of the option menu.

See `IlvListGadgetItemHolder`.

Changing and Retrieving the Selected Item

To modify the selected item in an option menu, use the member function `IlvOptionMenu::setSelected`. To retrieve the index of the selected item, use `IlvOptionMenu::whichSelected`.

Localizing Option Menus

Option menus can be localized.

See *Localizing a Gadget* on page 211.

Event Handling and Callbacks

When the user selects a new item in the menu, either by pointing on it with the mouse or by using the arrow keys, the virtual member function `IlvOptionMenu::doIt` is called. It can be overridden in a subclass of the option menu when necessary.

Its default implementation invokes the Main callback of the option menu.

See *Associating a Callback with a Gadget* on page 210.

Using `IlvPasswordField`

The `IlvPasswordField` class is a special text field for entering passwords. A special character replaces the characters that you type in the field so that the password remains secret. `IlvPasswordField` is a subclass of `IlvTextField`.

To retrieve the text entered by the user, use the `IlvTextField::getLabel` member function. To modify the character typed in place of the real text, call `IlvPasswordField::setMaskChar`.

Using IlvScrollBar

The class `IlvScrollBar` defines a rectangular area with two arrows and a slider used for scrolling through a window. This rectangular area is called a scrollbar.



Figure 11.17 A Scrollbar

This section covers the following topics:

- ◆ *Setting the Scrollbar Values*
- ◆ *Setting the Scrollbar Orientation*
- ◆ *Event Handling and Callbacks*

Setting the Scrollbar Values

A scrollbar is defined by the following values:

- ◆ Its current value.
- ◆ Its minimum and maximum values.
- ◆ The slider size.
- ◆ The increment, that is, the value added to or removed from the scrollbar current value when clicking the scrollbar arrows or when pressing the Left, Right, Up or Down keys.
- ◆ The page increment, that is, the value added to or removed from the current scrollbar value when clicking the areas between the slider and the arrows or when pressing the Page-Up or Page-Down keys.

The current value of the scrollbar can change within the minimum value and the (maximum - slider size) value.

Use the `IlvScrollBar::setValues` method to set the current value of the scrollbar and its minimum and maximum values.

Use the `IlvScrollBar::setIncrement` and `IlvScrollBar::setPageIncrement` methods to set the increment and the page increment.

Setting the Scrollbar Orientation

A slider can have four types of orientation, which are specified in the constructor. You can also change its orientation using `IlvScrollBar::setOrientation`. The orientation of the slider can be:

- ◆ `IlvLeft` Horizontal slider with minimum value on the left.
- ◆ `IlvRight` Horizontal slider with minimum value on the right.
- ◆ `IlvTop` Vertical slider with minimum value on the top.
- ◆ `IlvBottom` Vertical slider with minimum value on the bottom.

Event Handling and Callbacks

When the user drags the slider, thus causing the scrollbar value to change, the virtual member function `IlvScrollBar::drag` is called. This member function can be overridden in subclasses. Its default implementation invokes the member function `IlvScrollBar::valueChanged`.

`IlvScrollBar::valueChanged` is also called when the user clicks the scrollbar arrows or the area located between the slider and the arrows, or when the user presses the arrow keys or the Home and End keys. This virtual member function can be overridden in subclasses. Its default implementation invokes the Main callback associated with the scrollbar.

When the user releases the slider, after he dragged it, the virtual member function `IlvScrollBar::dragged` is called. This member function can be overridden in subclasses. Its default implementation invokes the secondary callback associated with the scrollbar.

See *Associating a Callback with a Gadget* on page 210.

Using `IlvSlider`

The class `IlvSlider` defines a rectangular area that contains a slider. When the user moves the slider, its value changes.

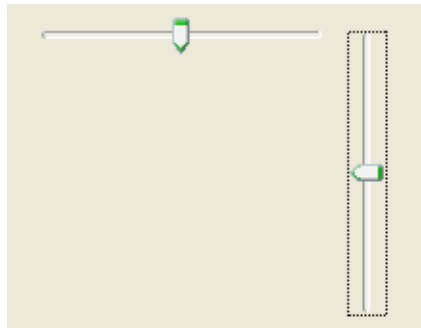


Figure 11.18 *Horizontal and Vertical Sliders*

This section covers the following topics:

- ◆ *Setting the Slider Values*
- ◆ *Setting the Slider Orientation*
- ◆ *Setting the Thumb Orientation*
- ◆ *Event Handling and Callbacks*

Setting the Slider Values

The class `IlvSlider` provides an easy way to modify a value between a range.

The slider is defined by the following values:

- ◆ Its current value.
- ◆ Its minimum and maximum values.
- ◆ The slider size.
- ◆ The page increment, that is, the value added to or removed from the slider current value when clicking the areas outside the slider or when pressing the Page Up or Page Down keys.

You can set the value and the range of the slider using the member function

`IlvSlider::setValues`. You can set the page increment with

`IlvSlider::setPageIncrement`.

Setting the Slider Orientation

A slider can have four types of orientation, which are specified in the constructor. You can also change its orientation using `IlvSlider::setOrientation`. The orientation of the slider can be:

- ◆ `IlvLeft` Horizontal slider with minimum value on the left.
- ◆ `IlvRight` Horizontal slider with minimum value on the right.
- ◆ `IlvTop` Vertical slider with minimum value on the top.
- ◆ `IlvBottom` Vertical slider with minimum value on the bottom.

Setting the Thumb Orientation

The thumb orientation can be also be set using the `IlvSlider::setThumbOrientation` method. However, this setting is not supported by all kinds of look-and-feel styles. For example, setting the thumb orientation has no effect when using the Motif look and feel.

The following illustration shows two sliders with different thumb orientation:



Figure 11.19 The Thumb Orientation of the Sliders

Event Handling and Callbacks

When the user drags the slider, clicks outside it, or presses the arrow keys, the Home or End keys, thus causing the slider value to change, the virtual member function

`IlvSlider::valueChanged` is called. This member function can be overridden in subclasses to perform a specific action. Its default implementation invokes the Main callback associated with the slider. Any changes made to the slider value call the slider callback.

See *Associating a Callback with a Gadget* on page 210.

Using IlvSpinBox

The class `IlvSpinBox` defines a composite gadget made up of two buttons and several fields of the type `IlvTextField` or `IlvNumberField`.

For text fields, you can define a list of predefined string values which the user can spin through using the buttons. For number fields, you can define a set of numeric values, within the specified value range, which the user can increment or decrement using the buttons.

You can also add graphic objects to a spin box.



Figure 11.20 A Spin Box

See *Using IlvNumberField* on page 247 and *Using IlvTextField* on page 264.

This section covers the following topics:

- ◆ *Adding and Removing Fields to a Spin Box*
- ◆ *Working with Text Fields*
- ◆ *Working with Numeric Fields*
- ◆ *Event Handling and Callbacks*

Adding and Removing Fields to a Spin Box

When created, a spin box has no fields; it is composed only of two arrow buttons. You can add one or more `IlvTextField` or `IlvNumberField` to a spin box. Note, however, that you can use a spin box that has no fields to increment or decrement a value in your application.

Adding Fields

To add a field to a spin box, you can use either one of these two member functions, depending on the type of values you want to display (character strings or numbers).

```
void addField(IlvTextField* field,
             const char** values,
             IlUShort count,
             IlUShort pos,
             IlBoolean loop,
             IlUShort at = 0,
             IlBoolean redraw = IlFalse);
```

The `values` parameter holds the string values that you will spin through. The `count` parameter specifies the number of strings in `values`.

```
void addField(IlvNumberField* field,
             IlDouble value,
             IlDouble increment,
             IlBoolean loop,
             IlUShort at = 0,
             IlBoolean redraw = IlFalse);
```

When you add a numeric field to a spin box, the buttons allow you to change the value of the numeric field within the value range specified by the numeric field itself (see *Using IlvNumberField* on page 247).

The `value` parameter is the initial value of the field. The `increment` parameter specifies the value that is added to or removed from the value of the numeric field when the user clicks the Increment or Decrement buttons.

If the `loop` parameter is set to `IlTrue`, the spin box returns to the first value when the user tries to increment the last value, and to the last value when the user tries to decrement the first value.

The `at` parameter lets you insert the field at a specific location in the spin box.

Here is a short example (`spinbox` is a pointer to an `IlvSpinBox` object):

```
const char* values[7] = {"Monday", "Tuesday", "Wednesday",
                       "Thursday", "Friday", "Saturday", "Sunday"};
spinbox->addField(new IlvTextField(display, "", IlvRect(0,0,10,10)),
                 values, 7, 0, IlTrue);
```


Note: The rectangle used for creating the `IlvTextField` has no meaning here. Also, you do not need to add the `IlvTextField` to a container because now it is managed by the spin box.

Removing Fields

To remove a field from an `IlvSpinBox`, use `IlvSpinBox::removeObject`. This member function also removes a graphic object added to the spin box as a decoration.

Adding Graphic Objects

You can add any graphic object to a spin box with the member function `IlvSpinBox::addObject`. Graphic objects appearing in a spin box serve as decorations and do not have any specific behaviors.

Working with Text Fields

If the field in a spin box is of type `IlvTextField`, you can retrieve its array of predefined strings with `IlvSpinBox::getLabels` and `IlvSpinBox::getLabelsCount`.

You can add a predefined string to a text field with `IlvSpinBox::addLabel` and remove it with `IlvSpinBox::removeLabel`.

You can set or retrieve the contents of a text field with these member functions:

```
const char* getLabel(IlvTextField* field) const;
void setLabel(IlvTextField* field,
              const char* label,
              IlBoolean redraw = IlFalse);
void setLabel(IlvTextField* field,
              IlUShort index,
              IlBoolean redraw = IlFalse);
```

Working with Numeric Fields

If the field in a spin box is of type `IlvNumberField`, you can set the increment specified with `IlvSpinBox::setIncrement` and retrieve it with `IlvSpinBox::getIncrement`.

The increment is the value that is added to or retrieved from the field value when the user clicks the spin box buttons.

You can set and retrieve the numeric value of a field with these member functions:

```
IlDouble getValue(IlvNumberField* field,
                  IlBoolean& error) const;
IlBoolean setValue(IlvNumberField* field,
                  IlDouble value);
```

Event Handling and Callbacks

The class `IlvSpinBox` defines two callback types, `Increment` and `Decrement`, which you can access using the following:

```
static IlvSymbol* IncrementCallbackType();
static IlvSymbol* DecrementCallbackType();
```

These callbacks are invoked when the user clicks the `Increment` and `Decrement` buttons. The active field, if any, is incremented/decremented just before the callback is invoked. The `Main` callback is called in both cases.

See *Associating a Callback with a Gadget* on page 210.

Using IlvStringList

The class `IlvStringList` displays a list of gadget items of the class `IlvGadgetItem`, or of a subclass. `IlvStringList` is a subclass of `IlvScrolledGadget` and `IlvListGadgetItemHolder`.

String lists can store up to 32767 items and can be composed of labels, bitmaps, or graphic objects (class `IlvGraphic`), and support scrollbars.

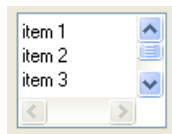


Figure 11.21 A String List

This section covers the following topics:

- ◆ *Manipulating String List Items*
- ◆ *Customizing the Appearance of String List Items*
- ◆ *Displaying Tooltips*
- ◆ *Localizing String List Items*
- ◆ *Handling Events and Callbacks*

Manipulating String List Items

Member functions for manipulating string list items are defined in the base class `IlvListGadgetItemHolder`.

Customizing the Appearance of String List Items

In addition to the graphic features of gadget items (see the class `IlvGadgetItem`), the class `IlvStringList` offers several ways to customize the global display of its items:

- ◆ *Defining Item Height*
- ◆ *Showing Label and Picture*
- ◆ *Setting Label and Picture Position*
- ◆ *Setting the Label Alignment*
- ◆ *Choosing a Selection Mode*

Defining Item Height

By default, items in a string list can have different heights. You can however choose to display all the items with the same height using the member function `IlvStringList::setDefaultItemHeight`.

Showing Label and Picture

Pictures in a string list can be shown or hidden with the member function `IlvStringList::showPicture`.

Similarly, labels in a string list can be displayed or hidden with the member function `IlvStringList::showLabel`.

By default, a string list displays both labels and pictures.

Note: You can override this global setting for a specific item with `IlvGadgetItem::showLabel` and `IlvGadgetItem::showPicture`.

Setting Label and Picture Position

You can change the position of the item labels relative to their pictures with `IlvStringList::setLabelPosition`.

By default, the label is placed to the right of the picture (`IlvRight`).

Note: You can override this global setting for a specific item with `IlvGadgetItem::setLabelPosition`.

Setting the Label Alignment

When the label position is `IlvRight` (the default value), and when only certain items are using a picture and a label, you may want all the labels to be left-aligned, as illustrated in Figure 11.21.

By default, item labels are automatically aligned. When an item is modified, the list recomputes the new label alignment. Since this operation can be time-consuming, it is possible to disable the automatic alignment of labels using the member function `IlvStringList::autoLabelAlignment`.

You may also want to disable the automatic label alignment mode because you know the size of all your pictures. In this case, call `IlvStringList::setLabelOffset`.

For example, the following call will ensure that each label item will be displayed with a left margin of 30 pixels:

```
slist->setLabelOffset(30);
```

Choosing a Selection Mode

When a string list item is selected, it is highlighted. The `IlvStringList` class provides two different modes for displaying selected items:

- ◆ **Full selection mode** When this mode (the default) is set, the selection extends on the whole width of the string list.
- ◆ **Partial selection mode** When this mode is set, the selection extends to the item labels only.

These two modes are illustrated in the figure below:

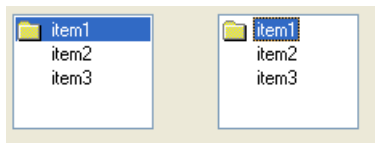


Figure 11.22 Full Selection Mode (Left) and Partial Selection Mode (Right)

You can switch from one mode to the other using `IlvStringList::useFullSelection`.

Displaying Tooltips

String lists can display tooltips when the mouse pointer is over partially visible items, provided that tooltips have been enabled with the member function `IlvStringList::useToolTips`.

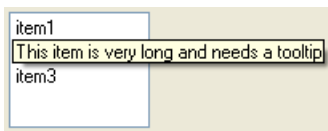


Figure 11.23 *Tooltip Displayed*

Note: *Tooltips work only if the partial selection mode is set. See [Choosing a Selection Mode](#) on page 260.*

Localizing String List Items

String list labels can be localized.

See [Localizing a Gadget](#) on page 211.

Handling Events and Callbacks

This section covers the following topics:

- ◆ *Selection Modes*
- ◆ *Selecting and Double-clicking a String List Item*
- ◆ *Editing a String List Item*
- ◆ *Dragging and Dropping a String List Item*

Selection Modes

There are two selection modes for string lists: single (or exclusive) selection and multiple selection.

In single selection mode, only one item can be selected at a time. This mode has two submodes:

- ◆ `IlvStringListSingleSelection`—You can select only one item at a time.
- ◆ `IlvStringListSingleBrowseSelection`—This mode is similar to the previous one except that clicking the selected item cancels the selection.

In multiple selection mode, several items can be selected at the same time. This mode has three submodes:

- ◆ `IlvStringListBrowseSelection`—You can select several items at the same time either by clicking them or dragging the mouse. Similarly, you can deselect several items by clicking them or by dragging the mouse with the middle button.
- ◆ `IlvStringListMultipleSelection`—Clicking an item selects it or cancels the selection.

- ◆ `IlvStringListExtendedSelection`—You can extend the selection using the Shift or the Control key.

To modify the selection mode in effect, use the member function `IlvStringList::setExclusive`. To change the submode, use `IlvStringList::setSelectionMode`. In multiple selection mode, you can set a limit to the number of items that can be selected with the member function `setSelectionLimit`.

Selecting and Double-clicking a String List Item

When the user double-clicks on a string list item, the Main callback is invoked. When the user selects an item or cancels the selection, the Select callback is called. To set this callback, use the member function `IlvStringList::setSelectCallback`.

See *Associating a Callback with a Gadget* on page 210.

To control selecting and double-clicking items in a list, you can redefine the following member functions in subclasses: `IlvStringList::select` (for selecting an item), `IlvStringList::unselect` (for cancelling the selection), or `IlvStringList::doIt` (for double-clicking on an item).

If you want to change the selection by coding, you can use `IlvStringList::setSelected`.

Editing a String List Item

Items in a string list can be edited. See *Finding Gadget Items* on page 284.

Dragging and Dropping a String List Item

The `IlvStringList` class provides an easy-to-use drag-and-drop mechanism. See *Dragging and Dropping Gadget Items* on page 286.

Using IlvText

The class `IlvText` defines a multiline text editor. Since `IlvText` is a subclass of `IlvScrolledGadget`, the text editor has scrollbars.



Figure 11.24 A Multiline Text Editor

The class `IlvText` provides a large number of member functions for setting or retrieving text, and for switching between the edit and read-only modes.

For details about handling scrollbars, see the base class `IlvScrolledGadget`.

This section covers the following topics:

- ◆ *Setting and Retrieving Text*
- ◆ *Event Handling*

Setting and Retrieving Text

You can specify the content of an `IlvText` object with the virtual member function `IlvText::setText` and retrieve it with `IlvText::getText`.

You can also set the text of a specific line with `IlvText::setLine` and retrieve it with `IlvText::getLine`.

You can add a line or remove a line with `IlvText::addLine` and `IlvText::removeLine`.

The class `IlvText` has many other helpful methods to set or retrieve several lines.

Event Handling

This section covers the following topics:

- ◆ *The check Method*
- ◆ *Keyboard Shortcuts*

The check Method

Each time the user types a regular ASCII character in a text gadget, the virtual member function `IlvText::check` is called. Its default implementation removes the selected text and adds the character that the user enters at the current cursor location.

Keyboard Shortcuts

The following table lists the keyboard shortcuts that can be used with text fields:

Key	Behavior
Home or Ctrl+A	Moves the cursor to the beginning of the line.
Meta <	Moves the cursor to the beginning of the text.
Meta >	Moves the cursor to the end of the text.
End or Ctrl+E	Moves the cursor to the end of the line.

Key	Behavior
Left arrow key or Ctrl+B	Moves the cursor left one character.
Right arrow key or Ctrl+F	Moves the cursor right one character.
Up key or Ctrl+P	Moves the cursor up one line.
Down key or Ctrl+N	Moves the cursor down one line.
Page Up	Moves the cursor one page up.
Page Down	Moves the cursor one page down.
Ctrl+K	Removes the text after the cursor.
Del or Ctrl+D	Removes the character after the cursor.
Back Space or Ctrl+H	Removes the character before the cursor.
Ctrl+X	Cuts the selected text to the clipboard.
Ctrl+C	Copies the selected text to the clipboard.
Ctrl+V	Pastes text from the clipboard.
Ctrl+Insert (Windows®)	Copies the selected text to the clipboard.
Shift+Insert (Windows)	Pastes text from the clipboard.
Ctrl+Left, Ctrl+Right	Moves the cursor one word backward or forward.
Shift+Left, Shift+Right Shift+Up, Shift+Down	Extends the selection one character up, down, left or right.
Ctrl+Shift+Left, Ctrl+Shift+Right	Extends the selection one word to the left or to the right.
Shift+Home, Shift+End	Extends the selection to the beginning or end of the line.
Ctrl+Shift+Home Ctrl+Shift+End	Extends the selection to the beginning or end of the text.

Using IlvTextField

The class `IlvTextField` defines a one-line text editor that is used to edit a short character string.



Figure 11.25 *A Text Field*

This section covers the following topics:

- ◆ *Aligning Text*
- ◆ *Setting and Retrieving Text*
- ◆ *Localizing a Text Field*
- ◆ *Limiting the Number of Characters*
- ◆ *Event Handling and Callbacks*

Aligning Text

The text of an `IlvTextField` can be left-aligned (the default), right-aligned, or centered. To change the text alignment, use `IlvTextField::setAlignment`.

Setting and Retrieving Text

Use the member functions `IlvTextField::setLabel` and `IlvTextField::getLabel` to set and retrieve text. The class `IlvTextField` also contains a set of useful methods for setting or retrieving formatted text such as integer or float values:

- ◆ `getIntValue()` retrieves an integer value.
- ◆ `getFloatValue()` retrieves a float value.
- ◆ `setValue(IlvInt)` sets an integer value.
- ◆ `setValue(IlvFloat, const char* format)` sets a float value.

Subclasses of `IlvTextField` edit an integer, a float, a date, and a password.

See *Using IlvDateField* on page 237, *Using IlvNumberField* on page 247.

Localizing a Text Field

Text fields in read-only mode can be localized.

See *Localizing a Gadget* on page 211.

Limiting the Number of Characters

You can limit the number of characters that can be edited in a text field with `IlvTextField::setMaxChar`. When its parameter is set to `-1`, you can type as many characters as you want. This member function limits the number of characters that you can

type in a text field, but not the number of characters you can specify with `IlvTextField::setLabel`. See *Setting and Retrieving Text* on page 265.

Event Handling and Callbacks

This section covers the following topics:

- ◆ *The Validate Method and the Main Callback*
- ◆ *The Check Method*
- ◆ *The labelChanged Method*

The Validate Method and the Main Callback

When the user presses the Enter key in a text field, the `IlvTextField::validate` member function is called. This virtual method invokes the Main callback of the text field and moves the focus to the next gadget in the focus chain.

Setting the Main callback for a text field provides an easy way to validate it. You can set a Focus Out callback to validate the text field instead of the Main callback. In this case, the field is validated when it loses the focus.

See *Associating a Callback with a Gadget* on page 210 and *Focus Management* on page 204.

The Check Method

Each time the user types a regular ASCII character in a text field, the virtual `IlvTextField::check` member function is called. Its default implementation removes the selected text and adds the characters that the user enters at the current cursor location.

This method checks the maximum number of characters allowed (see *Limiting the Number of Characters* on page 265). As a consequence, when you redefine it, be sure to add a test (similar to the one shown below) to allow this mechanism to work.

The labelChanged Method

When the user modifies the content of a text field, the member function `IlvTextField::labelChanged` is called. Its default implementation invokes the Change callback.

To set this callback, use `IlvTextField::setChangeCallback`.

See *Associating a Callback with a Gadget* on page 210.

Keyboard Shortcuts

The following table lists the keyboard shortcuts that can be used with text fields:

Key	Behavior
Home or Ctrl+A	Moves the cursor to the beginning of the text.
End or Ctrl+E	Moves the cursor to the end of the text.
Left arrow key or Ctrl+B	Moves cursor left one character.
Right arrow key or Ctrl+F	Moves cursor right one character.
Ctrl+K	Removes the text after the cursor.
Ctrl+U	Removes the text before the cursor.
Del or Ctrl+D	Removes the character after the cursor.
Back Space or Ctrl+H	Removes the character before the cursor.
Ctrl+X	Cuts the selected text to the clipboard.
Ctrl+C	Copies the selected text to the clipboard.
Ctrl+V	Pastes text from the clipboard.
Ctrl+Insert (Windows)	Copies the selected text to the clipboard.
Shift+Insert (Windows)	Pastes text from the clipboard.
Ctrl+Left, Ctrl+Right	Moves the cursor one word backward or forward.
Shift+Left, Shift+Right	Extends the selection one character to the left or to the right.
Ctrl+Shift+Left, Ctrl+Shift+Right	Extends the selection one word to the left or to the right.
Shift+Home, Shift+End	Extends the selection to the beginning or the end of the text.

Using IlvToggle

The class `IlvToggle` defines toggle and radio buttons. Toggle and radio buttons are made up of a label and a marker that shows a state. State markers can be represented as a rectangle or a diamond. The class `IlvToggle` has a subclass, `IlvColoredToggle`, that implements a toggle button whose marker can have a color.



Figure 11.26 A Toggle Button

This section covers the following topics:

- ◆ *Changing the State and Color of a Toggle Button*
- ◆ *Toggle and Radio Button Styles*
- ◆ *Displaying a Bitmap on a Toggle Button*
- ◆ *Aligning and Positioning the Label*
- ◆ *Changing the Size of the State Marker*
- ◆ *Localizing a Toggle Button*
- ◆ *Associating a Mnemonic with a Toggle Button*
- ◆ *Handling Events and Callbacks*
- ◆ *Grouping Toggle Buttons in a Selector*

Changing the State and Color of a Toggle Button

The appearance of the state marker changes according to the state of the related toggle or radio button (on or off). To set the state of a toggle button, use the member function `IlvToggle::setState` and `IlvToggle::getState` to retrieve it.

To set the color of a colored toggle marker, use `IlvColoredToggle::setCheckColor` and `IlvColoredToggle::getCheckColor` to retrieve it.

Toggle and Radio Button Styles

The class `IlvToggle` can have two different shapes: a normal toggle button or a radio button.

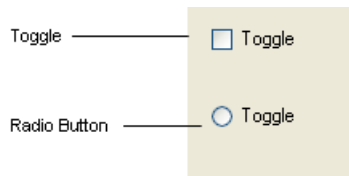


Figure 11.27 Various Styles of Toggle Buttons

To set the style for a toggle button as radio, use `IlvToggle::setRadio`.

Displaying a Bitmap on a Toggle Button

`IlvToggle` instances always display a label, even when they are explicitly requested to display a bitmap. If the toggle button is set to draw a bitmap and if its label is not empty, the `IlvToggle` instance is displayed with the label on top of the bitmap.

To display a bitmap on a toggle button, use the member function `IlvToggle::setBitmap`.

Aligning and Positioning the Label

The label of a toggle button can be placed to the right or to the left of the state marker. The label can also be left, right, or center-aligned in the space:

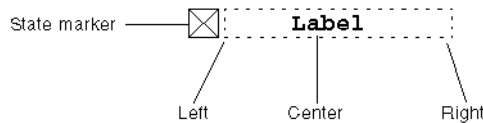


Figure 11.28 *Text Alignment in a Toggle Label*

To set the position of the label, use the member function `IlvToggle::setPosition`.

To set the alignment of the label, use the method `IlvToggle::setTextAlignment`.

Changing the Size of the State Marker

You can change the size of the state marker (that is, the height and width of its bounding box) with the member function `IlvToggle::setCheckSize`.

Giving a state marker size of 0, sets the state marker size to a default size.

Note: *When the Windows® look and feel is selected, changing the marker size has no effect.*

Localizing a Toggle Button

The label of toggle buttons can be localized.

See *Localizing a Gadget* on page 211.

Associating a Mnemonic with a Toggle Button

A toggle button can be associated with a mnemonic.

See *Associating a Mnemonic with a Gadget Label* on page 212.

Handling Events and Callbacks

When the user clicks on a toggle button, presses its associated mnemonic letter, or presses the Enter key or the space bar, the state of the button changes and the member function `activate` is called. This virtual member function calls the `Main` callback of the toggle button.

See *Associating a Callback with a Gadget* on page 210.

Grouping Toggle Buttons in a Selector

To create radio boxes, you can group toggle buttons into an `IlvSelector`. The `IlvSelector` class is a special kind of graphic set (`IlvGraphicSet`) that handles a unique selection among the objects it holds.

Two useful methods of the selector let you know what is selected:

```
IlvShort   whichSelected() const;
IlvGraphic* whichGraphicSelected() const;
```

Note: As the class `IlvSelector` is not a subclass of `IlvGadget` you must explicitly set the “Selector” interactor to have an interactive selector.

Source Program

```
#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/toggle.h>
#include <ilviews/graphics/selector.h>

static void QuitCallback(IlvView* top, IlvAny)
{
    IlvDisplay* display = top->getDisplay();
    delete top;
    delete display;
    IlvExit(0);
}

int main(int argc , char* argv[])
{
    IlvDisplay* display = new IlvDisplay("Demo", "", argc, argv);
    if (!display || display->isBad()) {
        IlvFatalError("Couldn't open display");
        delete display;
        IlvExit(-1);
    }

    IlvGadgetContainer* container =
        new IlvGadgetContainer(display,
                               "Demo",
                               "Demo",
                               IlvRect(0, 0, 100, 150));
```

```

container->setDestroyCallback(QuitCallback);

IlvSelector* selector = new IlvSelector;
IlvToggle* toggle;
toggle = new IlvToggle(display, IlvPoint(10, 10), "Toggle 1");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 50), "Toggle 2");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 90), "Toggle 3");
selector->addObject(toggle);

container->addObject("Selector", selector);

container->show();

IlvMainLoop();
return 0;
}

```

Creating the Selector

The selector is created by invoking its constructor:

```
IlvSelector* selector = new IlvSelector;
```

Then set the interactor:

```
selector->setInteractor(IlvInteractor::Get("Selector"));
```

Adding Toggle Buttons

Each toggle button is created and added to the selector with the `addObject` method:

```
IlvToggle* toggle;
toggle = new IlvToggle(display, IlvPoint(10, 10), "Toggle 1");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 50), "Toggle 2");
selector->addObject(toggle);
toggle = new IlvToggle(display, IlvPoint(10, 90), "Toggle 3");
selector->addObject(toggle);

```

Adding the Selector to its Container

```
container->addObject("Selector", selector);
```

Using IlvTreeGadget

An `IlvTreeGadget` is a gadget that displays a hierarchical list of items. Each item is an instance of the `IlvTreeGadgetItem` class, a subclass of `IlvGadgetItem`. The user may expand or collapse an item to display or hide its subitems.

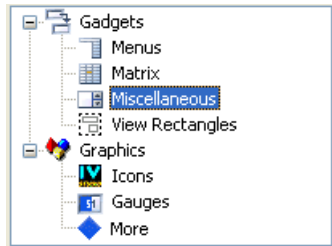


Figure 11.29 A Tree Gadget

The `IlvTreeGadget` class handles scrollbars. For details about handling scrollbars, see the base class `IlvScrolledGadget`.

This section covers the following topics:

- ◆ *Changing the Tree Hierarchy*
- ◆ *Navigating Through a Tree Hierarchy*
- ◆ *Changing the Characteristic of an Item*
- ◆ *Expanding and Collapsing a Gadget Item*
- ◆ *Changing the Look of the Tree Gadget Hierarchy*

Changing the Tree Hierarchy

The tree gadget has an invisible root item which you can retrieve using the `IlvTreeGadgetItemHolder::getRoot` member function.

Creating a Hierarchy

To create a hierarchical list of items, you first have to create the items that will be part of that list. Here are a few examples:

```
item1 = new IlvTreeGadgetItem("item1"); // Creates an item with a label.
item2 = new IlvTreeGadgetItem("item1", // Creates an item with a label
                               bitmap); // and a bitmap.
item3 = new IlvTreeGadgetItem(bitmap); // Creates an item with a bitmap.
item4 = new IlvTreeGadgetItem(graphic); // Creates an item with a graphic.
```

Once tree gadget items are created, you can arrange them as a tree structure in the following ways:

- ◆ Create tree gadget items as explained above and add them one by one to the tree gadget using the member function `IlvTreeGadget::addItem`.

- ◆ Create a complete new hierarchy and add it to the tree gadget in a single operation. This solution is far more efficient. To do so, create tree gadget items as explained above and add them as children using `IlvTreeGadgetItem::insertChild`. Then add the root item with `IlvTreeGadget::addItem`:

```
IlvTreeGadgetItem* item = new IlvTreeGadgetItem("New Item");
item->insertChild(new IlvTreeGadgetItem("Leaf 1"));
item->insertChild(new IlvTreeGadgetItem("Leaf2"));
tree->addItem(0 /* tree->getRoot() */, item);
```

Removing Tree Gadget Items

When you remove an item from a tree gadget, all its children are also removed from the tree.

To remove an item without destroying it, use

```
IlvTreeGadgetItemHolder::detachItem. To remove all the items at once, call
IlvTreeGadget::removeAllItems.
```

Moving Tree Gadget Items

You can move an item and all its children from its current parent item to a new parent item with `IlvTreeGadgetItemHolder::moveItem`.

Navigating Through a Tree Hierarchy

Once you have created a tree hierarchy, you can navigate in the tree using the member

```
functions IlvTreeGadgetItem::getParent,
IlvTreeGadgetItem::getFirstChild, IlvTreeGadgetItem::getNextSibling,
IlvTreeGadgetItem::getPrevSibling.
```

Changing the Characteristic of an Item

To change the visible characteristic of an `IlvTreeGadgetItem`, such as its label and bitmap, see the base class `IlvGadgetItem`.

You can specify whether the number of children of an item is known with `IlvTreeGadgetItem::setUnknownChildCount`. In this case, the tree gadget allows you to expand the item with the Expand button. This lets you have an Expand callback which is invoked even if the item does not have any subitems. You can then add items in the expand callback.

Expanding and Collapsing a Gadget Item

You can expand or collapse a gadget item either by clicking its Expand button or by double-clicking it. Expanding an item shows all its subitems; collapsing an item hides all its subitems. You can also perform the same operations using the following member functions `IlvTreeGadget::shrinkItem` and `IlvTreeGadget::expandItem`.

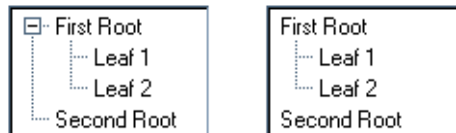
Changing the Look of the Tree Gadget Hierarchy

The way in which the `IlvTreeGadget` hierarchy is displayed can be customized to meet application requirements. To do so, use the following `IlvTreeGadget` member functions.

The lines that link items to their parents can be displayed or hidden with the `IlvTreeGadget::showLines` member function.



Lines can be drawn to connect the root item to its children using the `IlvTreeGadget::setLinesAtRoot` member function.



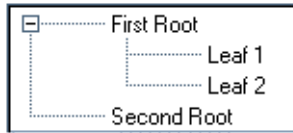
A line can be drawn to connect (or disconnect) the children of the root item using the `IlvTreeGadget::linkRoots` member function.



The buttons for expanding/collapsing may be set as visible or invisible using the `IlvTreeGadget::showButtons` member function.



You can define the indentation between an item and its parent using the `IlvTreeGadget::setIndent` member function.



Label and Picture Visibility

You can change the visibility of all the pictures in a tree gadget by calling this method:

```
void showPicture(IlBoolean value = IlTrue,
                IlBoolean redraw = IlTrue);
```

In the same way, you can change the visibility of all the labels in a tree gadget by calling this method:

```
void showLabel(IlBoolean value = IlTrue,
               IlBoolean redraw = IlTrue);
```

By default, the tree gadget displays both labels and pictures.

Note: You can override these global settings for a specific item through the API of the `IlvGadgetItem` class. For details, see the methods `IlvGadgetItem::showLabel` and `IlvGadgetItem::showPicture`.

Label and Picture Position

You may want to change the position of an item label relative to its picture. To do so, use the method:

```
void setLabelPosition(IlvPosition position,
                     IlBoolean redraw = IlTrue);
```

By default, the label is placed to the right of the picture (`IlvRight`).

Note: You can override these global settings for a specific item through the API of the `IlvGadgetItem` class. For details, see the `IlvGadgetItem::setLabelPosition` method.

Event Handling and Callbacks

The tree gadget has several predefined callbacks. Callbacks are always related to a particular item. To retrieve the item associated with the callback in your code, use the member function `IlvTreeGadget::getCallbackItem`.

Selection Modes

The tree gadget has two different selection modes:

- ◆ **Single selection mode** You can select only one item at a time.
- ◆ **Extended selection mode** You can select several items and expand the selection.

The selection modes are defined by the following type:

```
enum IlvTreeSelectionMode
{
    IlvTreeExtendedSelection = 0,
    IlvTreeSingleSelection   = 1
}
```

To modify the selection mode, use the following `IlvTreeGadget` member functions:

```
IlvTreeSelectionMode getSelectionMode() const;
void setSelectionMode(IlvTreeSelectionMode mode);
```

The Select Callback

When the user selects an item or cancels the selection, the `Select` callback is invoked. Its type can be retrieved with the member function `IlvTreeGadget::SelectCallbackType`. See *Associating a Callback with a Gadget* on page 210.

The Expand Callback

When the user expands an item, the `Expand` callback is invoked. Its type can be retrieved with the member function `IlvTreeGadget::ExpandCallbackType`. See *Associating a Callback with a Gadget* on page 210.

The Shrink Callback

When the user collapses an item, the `Shrink` callback is invoked. Its type can be retrieved with the member function `IlvTreeGadget::ShrinkCallbackType`. See *Associating a Callback with a Gadget* on page 210.

The Activate Callback

When the user double-clicks an item that has no subitems, the `Activate` callback is invoked. Its type can be retrieved with `IlvGadgetItemHolder::ActivateCallbackType`. See *Associating a Callback with a Gadget* on page 210.

Editing Tree Gadget Items

You can edit tree gadget items. See *Editing Gadget Items* on page 285.

Dragging and Dropping an Item

The `IlvTreeGadget` class provides an easy-to-use, drag-and-drop mechanism. See *Dragging and Dropping Gadget Items* on page 286.

Gadget Items

Most of the gadgets are composed of items, which are defined by the `IlvGadgetItem` class.

This chapter introduces you to gadget items and explains how to use them. It covers the following topics:

- ◆ *Introducing Gadget Items*
- ◆ *Using Gadget Items*
- ◆ *Gadget Item Holders*
- ◆ *List Gadget Item Holders*

Introducing Gadget Items

Gadget items are objects of the class `IlvGadgetItem`. Gadget items are gadget elements that can be represented by a label, a picture, or both. They can be dragged and dropped and be edited interactively. They can also display a tooltip and be localized. See *Localizing a Gadget* on page 211.

A gadget item does not implement behavior. Behavior is controlled by the gadget that manages it.

Gadget items are handled by the following gadget classes and their derived classes:

- ◆ IlvAbstractMenu
- ◆ IlvMatrix
- ◆ IlvMessageLabel
- ◆ IlvStringList
- ◆ IlvTreeGadget
- ◆ IlvNotebook

Figure 12.1 shows some of the gadgets that are composed of items. From left to right, you can see a button, a tree gadget, a string list, a pop-up menu, a tool bar, and an option menu.

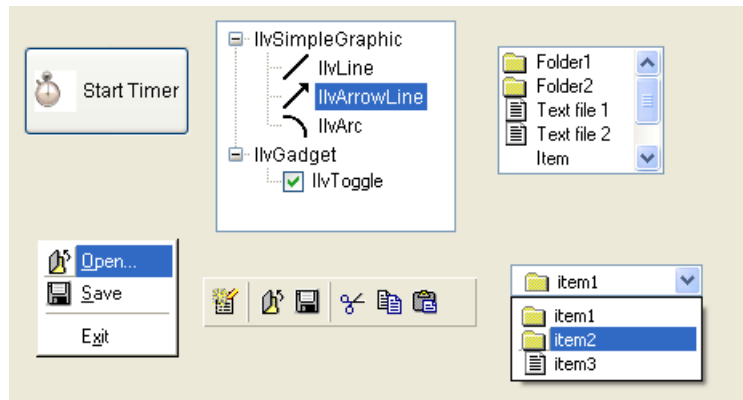


Figure 12.1 Gadgets Composed of Gadget Items

Using Gadget Items

This section covers the following topics:

- ◆ *Creating a Gadget Item*
- ◆ *Setting a Label*
- ◆ *Setting a Picture*
- ◆ *Specifying the Layout of a Gadget Item*
- ◆ *Nonsensitive Gadget Items*
- ◆ *Dynamic Types*
- ◆ *Using Palettes with Gadget Items*

◆ Drawing a Gadget Item

Creating a Gadget Item

A gadget item can be represented by a label, a picture, or both. The picture can be a bitmap or a graphic object. See *Setting a Label* on page 280 and *Setting a Picture* on page 281.

You define the way a gadget item appears when you create it. Here are a few examples:

```
item1 = new IlvGadgetItem("Item1"); // Creates an item with only a label.
item2 = new IlvGadgetItem("Item2", // Creates an item with a label
                           bitmap); // and a bitmap.
item3 = new IlvGadgetItem(bitmap); // Creates an item with a bitmap.
item4 = new IlvGadgetItem("Item 4", // Creates an item with a label
                           graphic); // and an IlvGraphic.
item5 = new IlvGadgetItem(graphic); // Creates an item with an IlvGraphic.
```

Setting a Label

A gadget item can be represented by a label. To associate a label with a gadget item, use the member function `IlvGadgetItem::setLabel`.

When a gadget item label extends over several lines, you can use

`IlvGadgetItem::setLabelAlignment` to specify whether the text should be aligned right, left, or be centered.

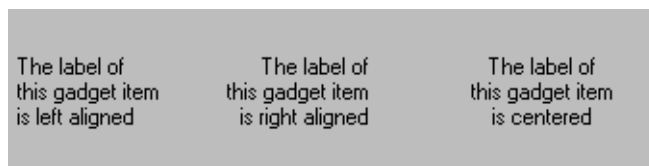


Figure 12.2 Message Labels with Various Alignments

Using the member function `IlvGadgetItem::setLabelOrientation`, you can also specify whether a gadget item label should be drawn horizontally (the default) or vertically.



Figure 12.3 Message Labels Displayed Vertically

Setting a Picture

A gadget item can include either an `IlvBitmap` or an `IlvGraphic` object. The following sections explain how gadget items handle these objects.

IlvBitmap Used as a Picture

A gadget item manages an array of bitmaps. Each bitmap in the array can be accessed by means of an index or a symbol name. You can retrieve the bitmap displayed by a gadget item from its bitmap array using `IlvGadgetItem::getCurrentBitmap`. This member function determines the displayed bitmap from the state of the gadget item. For example, if the gadget item is selected, the bitmap corresponding to the “selected” state is returned.

Below are the symbol names associated with the various gadget item states a bitmap can represent:

```
IlvGadgetItem::BitmapSymbol();           Sensitive state
IlvGadgetItem::SelectedBitmapSymbol();   Selected state
IlvGadgetItem::InsensitiveBitmapSymbol(); Nonsensitive state
IlvGadgetItem::HighlightedBitmapSymbol(); Highlighted state
```

To know how many bitmaps are associated with a gadget item, call the method `IlvGadgetItem::getBitmapCount`.

To set the bitmap that will be displayed by the gadget item when it is selected, call:

```
item->setBitmap(IlvGadgetItem::SelectedBitmapSymbol(), bitmap);
```

To retrieve the bitmap that is displayed when the gadget item is set to nonsensitive, call:

```
IlvBitmap* bitmap = item->getBitmap(IlvGadgetItem::InsensitiveBitmapSymbol());
```

IlvGraphic Used as a Picture

A gadget item can be represented by an `IlvGraphic` object. Use the member function `IlvGadgetItem::setGraphic` to associate a gadget item with a graphic object.

Specifying the Layout of a Gadget Item

You can define the position of a gadget item label relative to its picture using the member function `IlvGadgetItem::setLabelPosition`. For example, to place the label under the picture, call:

```
item->setLabelPosition(IlvBottom);
```

You can also fix the spacing between the label and the picture with `IlvGadgetItem::setSpacing`. For example, to set the spacing to 10 pixels, call:

```
item->setSpacing(10);
```

A gadget item can be any size. Its dimensions are automatically computed from its label, its picture, the label position, and the spacing between the label and the picture. To retrieve the size of a gadget item, use the following methods:

```
IlvDim width = item->getWidth();
IlvDim height = item->getHeight();
```

Note: *The width and height of a gadget item should not exceed 65535 pixels. The member functions `getWidth` and `getHeight` return 0 if the item is not managed by a gadget.*

To know the position of a label and a picture inside a gadget item, use the following member functions:

```
item->labelRect(rect, itembbox); // Puts the label bounding box of the item
                                // in rect when the item is drawn in itembbox.
item->pictureRect(rect, itembbox); // Puts the picture bounding box of the item
                                // in rect when the item is drawn in itembbox.
```

You can show or hide either the label or the picture that makes up a gadget item. To hide a gadget item label, use `IlvGadgetItem::showLabel` with its parameter set to `ILFalse`. If the gadget item contains no picture, it becomes invisible.

To make the picture visible, use the `IlvGadgetItem::showPicture` method with its parameter set to `ILTrue`.

Nonsensitive Gadget Items

By default, gadget items are sensitive, which means that they respond to user events. Calling the member function `IlvGadgetItem::setSensitive` with `ILFalse` as parameter lets you set a gadget item to nonsensitive. In this case, the gadget item appears dimmed on the screen and cannot be selected.

If only a sensitive bitmap is provided, the insensitive bitmap is computed automatically. If a nonsensitive bitmap is provided, this bitmap is used.

Dynamic Types

Gadget items are dynamically typed and can therefore be subclassed, saved, and read easily.

The following code lets you access class information:

```
IlvClassInfo* classInfo = item->getClassInfo();
```

To check the type of an item, use:

```
if (item->isSubtypeOf(IlvTreeGadgetItem::ClassInfo())) {
    // The item is an IlvTreeGadgetItem.
}
```

Using Palettes with Gadget Items

Several palettes are used to draw a gadget item:

- ◆ The palette returned by `IlvGadgetItem::getOpaquePalette` is used when the item is opaque.
- ◆ The palette returned by `IlvGadgetItem::getSelectionPalette` is used to draw the background of a selected gadget item.
- ◆ The palette returned by `IlvGadgetItem::getSelectionTextPalette` is used to draw the text of a selected gadget item.
- ◆ The palette returned by `IlvGadgetItem::getHighlightTextPalette` is used to draw the text of a highlighted gadget item.
- ◆ The palette returned by `IlvGadgetItem::getInsensitivePalette` is used to draw nonsensitive gadget items.
- ◆ The palette returned by `IlvGadgetItem::getNormalTextPalette` is used to draw the text of a gadget item that is not selected.

By default, a gadget item uses the palettes of its holder. You can, however, modify the palettes associated with a gadget item, thus making it possible to have gadget items with different palettes inside the same gadget.

To change the palettes assigned to a given gadget item, use the following member functions:

- ◆ `IlvGadgetItem::setNormalTextPalette`
- ◆ `IlvGadgetItem::setSelectionTextPalette`
- ◆ `IlvGadgetItem::setHighlightTextPalette`
- ◆ `IlvGadgetItem::setOpaquePalette`

Drawing a Gadget Item

To draw a gadget item, the virtual member function `IlvGadgetItem::draw` is called. You can override it in a subclass to customize the way a gadget item is drawn.

Gadget Item Holders

Gadget item holders are objects of the class `IlvGadgetItemHolder`, an abstract class for managing gadget items. A gadget item cannot compute its size and be drawn if it is not linked to a gadget item holder. Usually, you do not have to link a gadget item with its holder since this operation is carried out by the managing gadget automatically.

Gadget items are described in *Using Gadget Items* on page 279.

This section covers the following topics:

- ◆ *Gadget Item Features*
- ◆ *Finding Gadget Items*
- ◆ *Redrawing Gadget Items*
- ◆ *Creating Gadget Items*
- ◆ *Editing Gadget Items*
- ◆ *Dragging and Dropping Gadget Items*

Gadget Item Features

When global operations have to be performed on gadget items, it is a lot more convenient to call the corresponding functions on the holder than on the gadget items themselves. For example, it might be tedious to call:

```
item->showPicture(IfFalse);
```

for each item in a list whose picture you want to hide.

For this reason the gadget item inherits from certain features of its holder. If a given feature is not redefined at the gadget item level, the gadget item will get it from its holder. This is the case for the editable state, label and picture visibility, label position, and label orientation.

For example, if you want to hide all the pictures of a toolbar (`IlvToolBar` is a subclass of `IlvGadgetItemHolder`), just call:

```
toolbar->showPicture(IfFalse);
```

Then if you want to override this choice for the 4th item in the tool bar and show its picture, call:

```
toolbar->getItem(3)->showPicture(IfTrue);
```

Finding Gadget Items

The member function `IlvGadgetItemHolder::getItemByName` lets you find an item from its name. This method is particularly useful when searching for an item that is part of a tree structure (`IlvAbstractMenu` or `IlvTreeGadget`).

Redrawing Gadget Items

When you change the graphical representation of a gadget item using one of the `IlvGadgetItem` member functions, the gadget item is automatically redrawn.

In the following example, calling `setLabel` redraws the area of the list that was modified:

```
IlvStringList* list = ...
list->getItem(0)->setLabel("First Item");
```

If you want to apply several graphical representation changes at the same time, you can use the redraw mechanism of the gadget item holder, as shown below:

```
IlvStringList* list = ...
list->initReDrawItems();
list->getItem(0)->setLabel("First Item");
list->getItem(1)->setLabel("Second Item");
list->reDrawItems();
```

The redraw operation is executed only when the `IlvGadgetItemHolder::reDrawItems` method is called.

Creating Gadget Items

The `IlvGadgetItemHolder` class contains a method for creating an item from a specified label, bitmap, or `IlvGraphic` object:

```
virtual IlvGadgetItem* createItem(const char* label,
                                IlvGraphic* g = 0,
                                IlvBitmap* bitmap = 0,
                                IlvBitmap* sbitmap = 0,
                                IlBoolean copy = IlTrue) const;
```

This method creates an `IlvGadgetItem` object using the label, graphic, or bitmap passed as a parameter. It can be overridden in subclasses of `IlvGadgetItemHolder` to return a subclass of `IlvGadgetItem`. This is the case for the tree gadget, where `createItem` has been redefined to return an instance of `IlvTreeGadgetItem`.

Editing Gadget Items

The `IlvGadgetItemHolder` class supports gadget item editing for the following gadgets classes: `IlvMatrix`, `IlvStringList`, and `IlvTreeGadget`.

Enabling Editing

To make a gadget item editable, you must call the member function `IlvGadgetItem::setEditable` with `IlTrue` as its parameter. You can also enable editing at the level of the managing gadget with the either `IlvMatrix::allowEdit`, `IlvStringList::setEditable`, or `IlvTreeGadget::setEditable` depending on which class the gadget item belongs to.

For example, the following code allows editing for all the gadget items in the string list except the second item (specified by the index number 1).

```
slist->setEditable(IlTrue);
slist->getItem(1)->setEditable(IlFalse);
```

Note: You can also enable editing from the Start Edit callback. See *Controlling Editing* on page 286.

Editing a Gadget Item

Once editing has been enabled, you can edit a gadget item interactively either by clicking it or by pressing the F2 key after it has been selected. You can also edit an item by code using the member function `IlvGadgetItem::edit`.

Controlling Editing

When an item is being edited, two callbacks are invoked:

- ◆ Start Edit Item is called at the beginning of the editing process. To set a Start Edit Item callback, use the symbol returned by the member function `IlvGadgetItemHolder::StartEditItemCallbackType`.
You can cancel the operation by setting the item to noneditable inside this callback.
- ◆ End Edit Item is called at the end of the editing process. To set a End Edit Item callback, use the symbol returned by the member function `IlvGadgetItemHolder::EndEditItemCallbackType`.
You can cancel the editing of an item by pressing the Escape key. In this case, the End Edit Item callback is not invoked.

See “Callbacks” in *Graphic Objects*“.

Dragging and Dropping Gadget Items

The `IlvGadgetItemHolder` class implements the drag-and-drop functionality. Only instances of `IlvMatrix`, `IlvStringList`, and `IlvTreeGadget` support the drag-and-drop feature for gadget items.

Enabling Drag-and-Drop

To enable the drag-and-drop functionality for gadget items, you must call one of the following member functions with `IlTrue` as parameter: `IlvMatrix::allowDragDrop`, `IlvStringList::allowDragDrop`, or `IlvTreeGadget::allowDragDrop`.

Controlling Drag-and-Drop

When the drag-and-drop functionality is enabled, you can drag a gadget item from its current location and drop it anywhere. The following callbacks are invoked:

- ◆ Start Drag Item is called at the beginning of a drag-and-drop event. To set this callback, use the symbol returned by `IlvGadgetItemHolder::StartDragItemCallbackType`.

You can cancel the operation by calling the member function

`IlvGadgetItemHolder::setDraggedItem` with 0 as parameter from this callback.

- ◆ `Drag Item` is called each time the mouse is moved. To set this callback, use the symbol returned by `IlvGadgetItemHolder::DragItemCallbackType`.
- ◆ `End Drag Item` is called when a dragged item is dropped anywhere in the workspace. To set this callback, use the symbol returned by `IlvGadgetItemHolder::EndDragItemCallbackType`.

During a drag-and-drop operation, you can retrieve the dragged item using

`IlvGadgetItemHolder::getDraggedItem`. You can also change the ghost image of the item that is being dragged. By default, the ghost image is the dragged item drawn in XOR mode. To use a ghost image of your own, call

`IlvGadgetItemHolder::setDraggedImage` from the `Start Drag Item` or the `Drag Item` callback.

List Gadget Item Holders

List gadget item holders are specific types of gadget item holders for managing lists of gadget items. The class `IlvListGadgetItemHolder` is the base class of all the gadgets that handle gadget item lists, such as string lists and menus.

For information on gadget item holders, see *Gadget Item Holders* on page 283.

This section covers the following topics:

- ◆ *Modifying a List*
- ◆ *Accessing Items*
- ◆ *Sorting a List*

Modifying a List

All the member functions that modify a list redraw the modified area automatically. If you want to make several changes to a list without redrawing the area every time a modification is made, you can use the redraw mechanism of the `IlvGadgetItemHolder` class.

For details, see *Redrawing Gadget Items* on page 284.

Adding an Item to a List

Several member functions for adding items to a list are available. The most important one is the `IlvListGadgetItemHolder::insertItem`. This member function inserts an item inside a list at the specified position. Other methods, such as `addLabel` and `insertLabel`, call the `insertItem` method after they have created the item using the

`IlvGadgetItemHolder::createItem` method. For details, see *Creating Gadget Items* on page 285.

For example:

```
IlvStringList* list = ....
list->insertLabel("Label 1");
```

is equivalent to:

```
IlvStringList* list = ....
IlvGadgetItem* item = list->createItem("Label 1");
list->insertItem(item);
```

Changing All the Items in a List

You may want to change all the items in a list at once. To do this, use the `IlvListGadgetItemHolder::setItems` method. Using this method is more efficient than adding items one by one.

Here is an example of how to use the method `IlvListGadgetItemHolder::setItems`:

```
IlvUShort count = 3;
IlvGadgetItem** items = new IlvGadgetItem*[count];
items[0] = new IlvGadgetItem("item0");
items[1] = new IlvGadgetItem("item1");
items[2] = new IlvGadgetItem("item2");
IlvStringList* list = ...
list->setItems(items, count);
delete [] items;
```

Note that the items are not copied and that the `items` array is not used by the holder, and therefore needs to be deleted.

Other member functions, such as the `setLabels` methods, can be used to change a whole list. All these functions call the member function

```
IlvListGadgetItemHolder::setItems.
```

Removing an Item From a List

To remove an item from a list, use the `IlvListGadgetItemHolder::removeItem` member function.

Removing all Items

To remove all items from a list, use the `IlvListGadgetItemHolder::empty` member function.

Accessing Items

To know the number of items managed by a list gadget item holder, use the `IlvListGadgetItemHolder::getCardinal` member function.

To retrieve an item using its position in the list, use the `IlvListGadgetItemHolder::getItem` member function.

To find the position of an item in its holder, use the `IlvListGadgetItemHolder::getIndex` member function.

You can also find an item knowing its label using the `IlvListGadgetItemHolder::getPosition` member function.

Sorting a List

You can sort a list using the member function `IlvListGadgetItemHolder::sort`, which takes a comparison function as a parameter. If you do not provide your own comparison function, the virtual member function `IlvListGadgetItemHolder::compareItems` is used. This method simply uses the `strcmp` function to compare two strings and returns the result of the comparison.

If you want to use another function, you can either specify it in the call to `sort` or redefine the `compareItems` member function in your subclass of `IlvListGadgetItemHolder`.

The following is an example of a list compare function that sorts items in descending order:

```
int MyCompareFunction(const char* string1,
                    const char* string2,
                    IlvAny,
                    IlvAny)
{
    return -strcmp(string1, string2);
}
```

Menus, Menu Bars, and Toolbars

The IBM® ILOG® Views Gadgets library provides classes for creating menus and toolbars and for handling menu items.

This chapter covers the following topics:

- ◆ *Introducing Menus, Menu Bars, and Toolbars*
- ◆ *Menus and Menu Items*
- ◆ *Pop-up Menus*
- ◆ *Menu Bars and Toolbars*

Introducing Menus, Menu Bars, and Toolbars

Menus provide the user with a set of commands. When the user selects a menu or toolbar entry, a specific action is performed immediately or a dialog box is displayed in which the user is required to supply additional information before the action can be carried out. Menus can be attached to menu bars or toolbars or can be stand-alone.

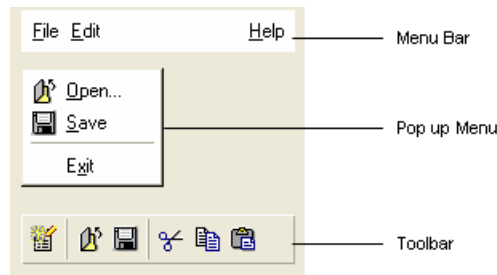


Figure 13.1 Menus

Menus and Menu Items

This section introduces the classes for defining menus and menu items. It covers the following topics:

- ◆ *Using IlvAbstractMenu*
- ◆ *Using IlvMenuItem*

Using IlvAbstractMenu

The class `IlvAbstractMenu`, a subclass of `IlvGadget`, defines a common interface for menu bars, toolbars, and pop-up menus. `IlvAbstractMenu` also inherits from the class `IlvListGadgetItemHolder`, which handles lists of gadget items. `IlvAbstractMenu` handles a list of `IlvMenuItem` objects, a subclass of `IlvGadgetItem`.

Manipulating Menu Items

Member functions for manipulating menu items are defined in the class `IlvListGadgetItemHolder`.

Callbacks

When the user highlights a menu item, the `Highlight` callback is invoked. This callback allows actions to take place according to the user selection. For example, the `Highlight` callback can be used to display a small help message when the user highlights an item in a pop-up menu.

You can set a `Highlight` callback with the symbol returned by the member function `IlvAbstractMenu::HighlightCBSymbol`.

Here is an example of `Highlight` callback that simply writes the index of the highlighted item:

```
static void
```

```

Highlight(IlvGraphic* g, IlvAny any)
{
    // Highlighted item position.
    IlvShort pos = *(IlvShort*)any;
    IlvAbstractMenu* menu = (IlvAbstractMenu*)g;
    if (pos != -1)
        IlvPrint("Item %d highlighted", pos);
    else
        IlvPrint("No item highlighted");
}

```

Note: Once it has been cast to `IlvShort`, the value of the `any` parameter is the position of the highlighted menu item, or `-1` if no item is highlighted.

Handling Events

The class `IlvAbstractMenu` includes the following virtual member functions that you can redefine in subclasses:

- ◆ `IlvAbstractMenu::isSelectable` specifies whether a menu item can be selected.
- ◆ `IlvAbstractMenu::selectNext` and `IlvAbstractMenu::selectPrevious` return the next or previous selectable item when the user moves in the menu using the arrow keys.
- ◆ `IlvAbstractMenu::select` and `IlvAbstractMenu::unSelect` are called when the specified item is selected or deselected.

Using `IlvMenuItem`

Menu bars, toolbars, and pop-up menus are composed of several entries, called menu items. Menu items are implemented by the `IlvMenuItem` class, a subclass of `IlvGadgetItem`. They can display a label, a bitmap, or any `IlvGraphic` object. See Chapter 12, *Gadget Items*.

Creating Menu Items

The following code sample creates three menu items: one with a label, one with a bitmap, and one with an `IlvGraphic` object.

```

item1 = new IlvMenuItem("item1");           // Creates an item with a label.
item2 = new IlvMenuItem(bitmap);           // Creates an item with a bitmap.
item3 = new IlvMenuItem(graphic);          // Creates an item with a graphic.

```

A menu item can also be used as a separator. A separator is a line that divides a group of commands represented by menu items in a menu.

```

item4 = new IlvMenuItem();                  // Creates a separator.

```

You can check whether an item is a separator or not using the `IlvMenuItem::getType` member function, as follows:

```
if (item->getType() == IlvSeparatorItem) {  
    ...  
}
```

Attaching a Submenu to a Menu Item

Any menu item that is not a separator can display a submenu. To attach a submenu to a menu item, use the member function `IlvMenuItem::setMenu`. When the menu item belongs to a pop-up menu, a small arrow next to it indicates that it provides access to a submenu.

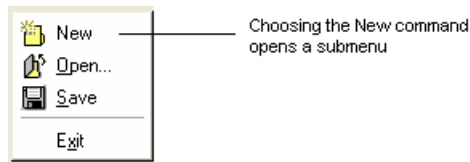


Figure 13.2 New Menu Item with a Submenu

Associating a Callback with a Menu Item

When the user selects a menu item, its associated callback is invoked to perform an action. Each menu item can have a specific callback.

To attach a callback to a menu item, use one of the following member functions:

- ◆ `item->setCallback(myCallback);`

where `myCallback` is a function that might be described like this:

```
static void  
myCallback(IlvGraphic* g, IlvAny data)  
{  
    ...  
}
```

The `g` parameter is the holder of the item that triggers the callback, that is, an instance of a subclass of `IlvAbstractMenu`. The `data` parameter is the client data of the menu item which you can install with the member function `IlvGadgetItem::setClientData`.

Of course, it is useless to set a callback to a menu item separator or to a menu item that has a submenu, as these callbacks will never be called.

- ◆ `item->setCallbackName("myCallback");`

In this case, the callback name "myCallback" must be registered with the container that holds the menu.

If a menu item does not have a callback, the Main callback associated with the menu, if any, is invoked. This allows you to perform the same action for each item of the menu. See *Associating a Callback with a Gadget* on page 210.

Associating Mnemonics with Menu Items

You can associate a mnemonic letter with a menu item. Pressing the modifier key (Alt on PCs, and Meta on UNIX) and the mnemonic letter associated with a menu or toolbar item displays the attached pop-up menu. When a menu is open, pressing the mnemonic letter selects the corresponding command in that menu, that is, triggers the Menu Item callback.

See *Associating a Mnemonic with a Gadget Label* on page 212.

Associating Accelerators with Menu Items

A pop-up menu item can be associated with an accelerator. An accelerator is a combination of a letter key with a modifier key. When the user presses the key combination, the Menu Item callback is directly accessed without the corresponding menu being opened.

An accelerator is composed of two parts: a key combination and the accelerator itself. The key combination appears beside its associated menu item.

For example, if you want to assign the key combination Ctrl+A to a menu item, use the following code:

```
item->setAcceleratorText("Ctrl+A");  
item->setAcceleratorModifiers(0);  
item->setAcceleratorKey(IlvCtrlChar('A'));
```

Pop-up Menus

A pop-up menu consists of a list of menu items laid out vertically. Pop-up menus are implemented with the class `IlvPopupMenu`, a subclass of `IlvAbstractMenu`. See *Using IlvAbstractMenu* on page 291 and *Using IlvMenuItem* on page 292.

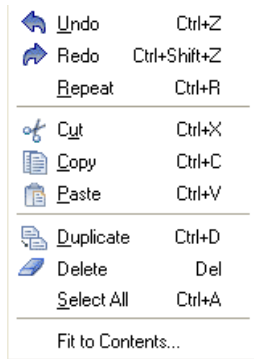


Figure 13.3 A Pop-up Menu

This section covers the following topics:

- ◆ *Aligning Item Labels in a Pop-up Menu*
- ◆ *Using Tear-Off Menus*
- ◆ *Using the Open Menu Callback*
- ◆ *Using Checked Menu Items*
- ◆ *Using Stand-alone Menus*
- ◆ *Using Tooltips in a Pop-Up Menu*

Aligning Item Labels in a Pop-up Menu

By default, the labels in a pop-up menu are automatically aligned as illustrated by the leftmost popup-menu in Figure 13.4. However, you can specify your own label offset with the member function `IlvPopupMenu::setLabelOffset`.

The middle image represents a pop-up menu for which the default alignment mode has been deactivated and no specific label offset has been defined. The rightmost image shows a pop-up menu aligned with a label offset of 40 pixels.



Figure 13.4 Aligning Menu Items Labels

Using Tear-Off Menus

A pop-up menu can be torn off, which means that it can be detached from the menu bar and placed into a floating window. A tear-off menu is represented by a dashed line across its top border.

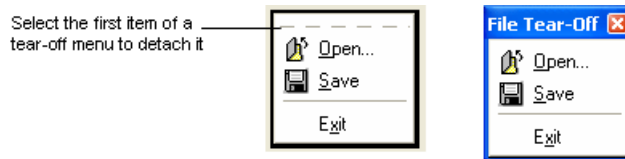


Figure 13.5 A Tear-Off Menu

You can tear off a pop-up menu by selecting its first item. The first item of a tear-off menu must be of the type `IlvTearOffItem`. To set a menu item as a tear-off item, use the `IlvMenuItem::setTearOff` member function.

Using the Open Menu Callback

Each time the user opens a pop-up menu, the Open Menu callback is invoked. This callback is particularly useful when you want items in the menu to change according to the state of the application, for example from “Save (Not Needed)” to “Save (Needed)” when there is something to save. The easiest way to achieve this is to set an Open Menu callback that verifies the state of the application and changes the item label accordingly.

You can set an Open Menu callback with the member function

`IlvPopupMenu::OpenMenuCallbackSymbol`. See *Associating a Callback with a Gadget* on page 210.

Using Checked Menu Items

Menu items in pop-up menus can have a small check mark appear beside them (a ✓ for the Microsoft® Windows® style or a small button for Motif®). Check marks are generally used with menu items that represent “on/off” options.

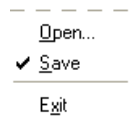


Figure 13.6 A Checked Menu Item

The best place to set a check mark for a menu item is the Open Menu callback. That way the check mark is always correct when the menu is opened. See *Using the Open Menu Callback* on page 296.

The check mark does not automatically disappear when you select a checked item, you must uncheck the item when necessary. To set a check mark for a menu item, use the member function `IlvMenuItem::setChecked`. Use `IlvMenuItem::isChecked` to know whether a menu item has a check mark.

Using Stand-alone Menus

Pop-up menus can be used either as submenus or as stand-alone menus. Most stand-alone menus are used as contextual menus, which appear when the user clicks in the workspace (generally with the right mouse button).

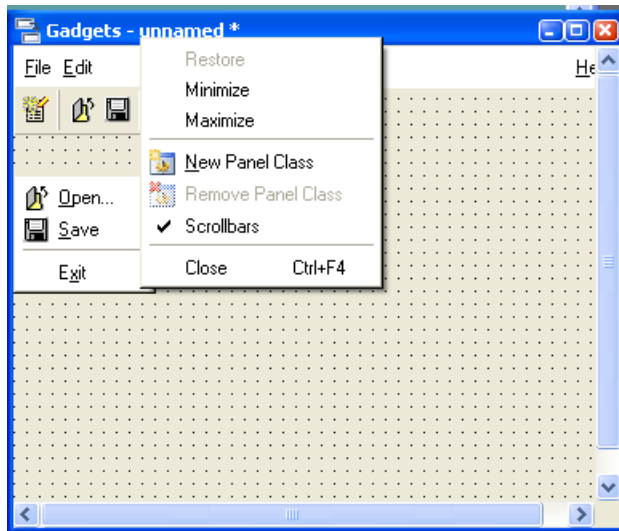


Figure 13.7 A Contextual Menu

To use a pop-up menu as a stand-alone menu, use the `IlvPopupMenu::get` member function.

When the user chooses an item from a contextual menu, the member function `IlvPopupMenu::doIt` is called.

Note: A contextual menu cannot be a submenu.

Using Tooltips in a Pop-Up Menu

The menu items in a pop-up menu can be associated with a tooltip. A tooltip is short explanatory text that is displayed when the user places the mouse over its associated menu item.

To set a tooltip for a menu item, use `IlvMenuItem::setToolTip`. To disable tooltips, call `IlvPopupMenu::useToolTips` with its parameter set to `IlFalse`.

Menu Bars and Toolbars

In IBM® ILOG® Views Gadgets, menu bars and toolbars are implemented by the classes `IlvMenuBar` and `IlvToolBar` respectively. Both these classes derive from `IlvAbstractBar`, a subclass of `IlvAbstractMenu`.

This section covers the following topics:

- ◆ *Using IlvAbstractBar*
- ◆ *Using IlvMenuBar and IlvToolBar*

Using IlvAbstractBar

`IlvAbstractBar` is an abstract class for managing the size and position of menu bar or toolbar items. See *Using IlvAbstractMenu* on page 291 and *Using IlvMenuItem* on page 292.

This section covers these topics:

- ◆ *Setting the Bar Orientation*
- ◆ *Constraining the Bar Geometry*
- ◆ *Notifying the Bar About Geometry Changes*
- ◆ *Setting the Default Item Size*
- ◆ *Aligning Items Flush-right*
- ◆ *Using Docking Features*

Setting the Bar Orientation

You can specify the orientation of the bar with the member function `IlvAbstractBar::setOrientation` and retrieve it with `IlvAbstractBar::getOrientation`.

The bar can be vertical, in which case menu items are arranged from top to bottom, or it can be horizontal, in which case items are arranged from left to right.



Figure 13.8 Vertical and Horizontal Toolbars

Constraining the Bar Geometry

You can constrain the bar geometry so that all its items are visible whatever its size with the member function `IlvAbstractBar::setConstraintMode`. When this member function is set to `IlTrue`, the bar is automatically resized to accommodate all its items. Items can be extended to several lines if necessary. To know whether the constraint mode is on, call `IlvAbstractBar::useConstraintMode`.

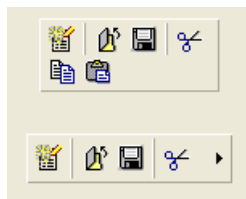


Figure 13.9 Constrained (Top) and Nonconstrained Toolbars (Bottom)

Notifying the Bar About Geometry Changes

When the constraint mode is on, the virtual member function `IlvAbstractBar::geometryChanged` is called if:

- ◆ Modifying the height of a vertical bar causes its width to change.
- ◆ Modifying the width of a horizontal bar causes its height to change.

Setting the Default Item Size

You can set a default size for all the items in a bar with the member function `IlvAbstractBar::setDefaultItemSize` and retrieve it with `IlvAbstractBar::getDefaultItemSize`. You can specify the spacing between two items in a bar with `IlvAbstractBar::setSpacing` and retrieve it with `IlvAbstractBar::getSpacing`.

Aligning Items Flush-right

You can align the last item in a bar with its right border with the member function `IlvAbstractBar::setFlushingRight`. Help menus, for example, are flush-right most of the time.

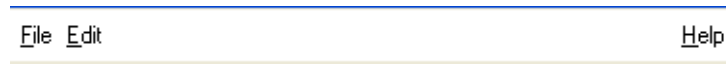


Figure 13.10 Help Menu Aligned Flush-right

Using Docking Features

You can dock and undock abstract bar objects. See *Using Docking Bars* on page 339.

Using IlvMenuBar and IlvToolBar

The classes `IlvMenuBar` and `IlvToolBar` define menu bars and toolbars.

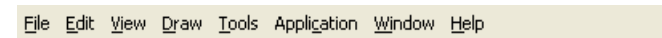


Figure 13.11 A Menu Bar

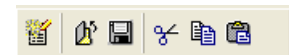


Figure 13.12 A Toolbar

These classes are very similar. The only difference is that `IlvToolBar` provides interactive features that the `IlvMenuBar` does not support, such as tooltips and gadgets.

Managing Gadgets in a Toolbar

You can use gadgets as toolbar items using the member function `IlvGadgetItem::setGraphic`. These gadgets are active, which means that they react to user events.

You can add a gadget to a toolbar with the member function `IlvListGadgetItemHolder::insertGraphic`.

Figure 13.13 shows a toolbar with a combo box.

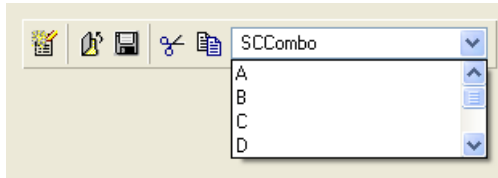


Figure 13.13 A Toolbar with a Gadget

When the user clicks a gadget in a toolbar, the gadget is given the focus and all keyboard events are directly sent to it. See *Focus Management* on page 204.

You can force the focus to be given to a specific item with `IlvToolBar::setFocusItem` and retrieve the gadget that has the focus with `IlvToolBar::getFocusItem`.

Using Tooltips in a Toolbar

Menu items in a toolbar can be associated with a tooltip. A tooltip is short explanatory text that is displayed when the user places the mouse over its associated menu item.

To set a tooltip for a menu item, use `IlvMenuItem::setToolTip`. To disable tooltips, call `IlvToolBar::useToolTips` with its parameter set to `IlFalse`.

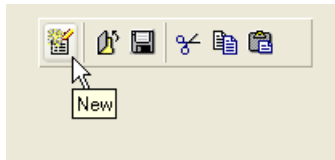


Figure 13.14 Tooltip Displayed

Matrices

The IBM® ILOG® Views Gadgets library provides classes for creating matrices.

This chapter covers the following topics:

- ◆ *Introducing Matrices*
- ◆ *Using IlvAbstractMatrix*
- ◆ *Using IlvMatrix*
- ◆ *Using IlvSheet*
- ◆ *Using IlvHierarchicalSheet*

Introducing Matrices

A matrix is a rectangular area made up of rows and columns that form a grid. The intersection of a row and a column forms a cell. A matrix can contain various matrix items, such as labels, numbers, graphic objects, gadgets, or gadget items. A matrix can have scrollbars.

The IBM® ILOG® Views Gadgets classes that implement matrices are `IlvAbstractMatrix`, `IlvMatrix`, `IlvSheet`, and `IlvHierarchicalSheet`.

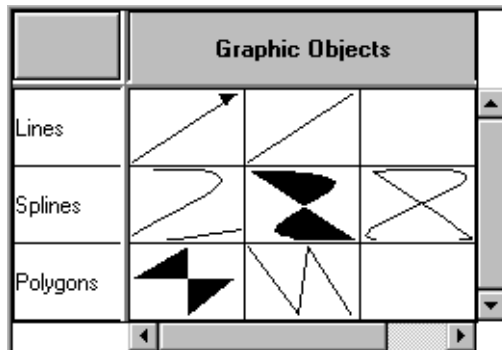


Figure 14.1 A Matrix

Using IlvAbstractMatrix

The class `IlvAbstractMatrix` is an abstract class for drawing matrices. Several of its member functions are virtual and must be redefined in subclasses. This class allows you to specify whether items should extend over several rows or columns, and also how many fixed rows and columns a matrix should contain. It also manages scrolling.

This section covers the following topics:

- ◆ *Subclassing `IlvAbstractMatrix`*
- ◆ *Drawing Items Over Multiple Cells*
- ◆ *Setting Fixed Rows and Columns*
- ◆ *Handling Events*

Subclassing `IlvAbstractMatrix`

The class `IlvAbstractMatrix` does not contain any values. It provides a set of pure virtual member functions that must be implemented in subclasses:

- ◆ `IlvAbstractMatrix::rows` and `IlvAbstractMatrix::columns` must return the number of rows and columns in the matrix.
- ◆ `IlvAbstractMatrix::rowSameHeight` and `IlvAbstractMatrix::columnSameWidth` must return `ILTrue` if all the rows and columns should have the same height and width.

- ◆ `IlvAbstractMatrix::getRowHeight` and `IlvAbstractMatrix::getColumnWidth` must return the height of each row and the width of each column. If `rowSameHeight` returns `IlTrue`, `getRowHeight(0)` returns the height of the rows and `getColumnWidth(0)` returns the width of the columns.
- ◆ `IlvAbstractMatrix::drawItem` draws an item in a matrix at the specified location defined by a row and a column number. This member function also specifies the bounding box of the matrix item and a clip rectangle.

Drawing Items Over Multiple Cells

You can have items extend to multiple rows and columns. To enable this feature, you must set the Boolean member value `_allowCellMode` to `IlTrue` in the `IlvAbstractMatrix` constructor. Also, you must redefine `IlvAbstractMatrix::cellInfo`. This member function specifies how many rows and columns the matrix item spans and the position of its top-left cell.

In the following example, the matrix item is defined to start at position (10,10) and to occupy five rows and five columns:

```
if ((colno >= 10) && (colno < 15) &&
    (rowno >= 10) && (rowno < 15))
{
    startcol = 10;
    startrow = 10;
    nbc col = 5;
    nbrow = 5;
}
else
    IlvAbstractMatrix::cellInfo(colno, rowno,
                                startcol, startrow,
                                nbc ol, nbrow);
```

Note: *Items extending over several rows and columns cannot overlap.*

When this member function is redefined, only the top-left cell is drawn (see `IlvAbstractMatrix::drawItem`). The rectangle passed to the `drawItem` member function encompasses all the rows and columns that the matrix item covers.

Setting Fixed Rows and Columns

You can specify that a number of rows and columns in a matrix remain fixed. Fixed rows and columns are always visible even when the user scrolls the matrix. Only the leftmost columns and the topmost rows can be fixed.

To have fixed rows or columns, use `IlvAbstractMatrix::setNbFixedRow` and `IlvAbstractMatrix::setNbFixedColumn`.

Handling Events

The class `IlvAbstractMatrix` does not define particular behaviors.

The member function `IlvAbstractMatrix::handleEvent` simply handles events related to scrollbars, if the matrix has scrollbars, and calls `IlvAbstractMatrix::handleMatrixEvent`.

If you want to implement a specific behavior for a matrix, you must redefine this member function in a subclass.

The following methods can help you write the behavior for your class:

```
virtual IlvBoolean  pointToPosition(const IlvPoint& p,
                                   IlvShort& colno,
                                   IlvShort& rowno,
                                   const IlvTransformer* t = 0) const;
```

This method returns, in `colno` and `rowno`, the location of the item which is under the point `p` when the matrix is displayed using the transformer `t`. The returned value is `IlvTrue` if there is an item at this location, or `IlvFalse` if there is none.

```
IlvBoolean rowBBox(IlvShort rowno,
                  IlvRect& rect,
                  const IlvTransformer* t = 0) const;
IlvBoolean columnBBox(IlvShort colno,
                     IlvRect& rect,
                     const IlvTransformer* t = 0) const;
IlvBoolean cellBBox(IlvShort colno,
                   IlvShort rowno,
                   IlvRect& rect,
                   const IlvTransformer* t = 0) const;
```

The above methods compute in `rect` the bounding box of a column, a row, or a cell when the matrix is drawn with the transformer `t`. The method returns `IlvTrue` if the item is visible (even partially), or `IlvFalse` if it is not.

To redraw a column, use the `IlvAbstractMatrix::invalidateColumn` method. To redraw a row, use the `IlvAbstractMatrix::invalidateRow` method.

Using IlvMatrix

A matrix is an instance of the `IlvMatrix` class, a subclass of `IlvAbstractMatrix`. A matrix is a rectangular grid made up of rows and columns, which can contain many different types of objects (labels, graphic objects, other gadgets, and so on). These objects, called matrix items, are of the class `IlvAbstractMatrixItem`.

This section covers the following topics:

- ◆ *Handling Columns and Rows*
- ◆ *Handling Matrix Items*
- ◆ *Handling Events*
- ◆ *Using Gadget Items in a Matrix*

Handling Columns and Rows

This section introduces the various operations you can perform on rows and columns:

- ◆ *Adding Rows and Columns*
- ◆ *Resizing Rows and Columns*
- ◆ *Setting the Automatic Fit-to-Size Mode*

Adding Rows and Columns

You can specify the number of rows or columns that a matrix will contain in the `IlvMatrix` constructor.

```
IlvMatrix(IlvDisplay* display,
          const IlvRect& rect,
          IlvShort nbc,
          IlvShort nbrow,
          IlvDim xgrid = IlvDefaultMatrixWidth,
          IlvDim ygrid = IlvDefaultMatrixWidth,
          IlvDim thickness = IlvDefaultGadgetThickness,
          IlvPalette* palette = 0);
```

Note: A matrix must have at least one row and one column.

You can add new columns and rows to a matrix with the `IlvMatrix::insertColumn` and `IlvMatrix::insertRow` member functions and remove them with `IlvMatrix::removeColumn` or `IlvMatrix::removeRow`.

You can modify the number of columns and rows in a matrix in one operation using the member function `IlvMatrix::reinitialize`.

Resizing Rows and Columns

The initial width of a column and height of a row are specified by the `xgrid` and `ygrid` parameters provided to the `IlvMatrix` constructor. When a matrix is created, its rows and columns all have the same dimensions that are indicated by these parameters. You can, however, modify the original settings with the `IlvMatrix::setXgrid` and `IlvMatrix::setYgrid` member functions, which let you set the width of each column and the height of each row, respectively. Also, you can change the size of each individual column or row with the member functions `IlvMatrix::resizeColumn` and

`IlvMatrix::resizeRow`. In this case, the global settings defined for the matrix are no longer used, and modifying their values will have no effect on the dimensions of the other rows and columns.

To revert to a matrix whose columns and rows are all of the same size, use `IlvMatrix::sameHeight` and `IlvMatrix::sameWidth`.

Setting the Automatic Fit-to-Size Mode

You can request that the dimensions of the columns and rows in a matrix be adjusted automatically when the matrix is resized with the member function

`IlvMatrix::autoFitToSize`. This feature does not apply when a matrix has scrollbars. When the “auto fit to size” mode is set, you can specify that only the width of the last column or the height of the last row be adjusted when the matrix is resized with `IlvMatrix::adjustLast`.

You can also recompute the size of all the columns and rows so that they fit into the matrix bounding box with `IlvMatrix::fitToSize`.

Handling Matrix Items

Matrix items are instances of subclasses of the class `IlvAbstractMatrixItem`. Matrix items can be selected and edited. Gadgets used as matrix items are active, meaning that they react to user input.

This section covers the following topics:

- ◆ *Predefined Matrix Item Classes*
- ◆ *Creating a New Subclass of Matrix Items*
- ◆ *Adding and Removing Matrix Items*
- ◆ *Redrawing Matrix items*
- ◆ *Aligning Matrix Items*
- ◆ *Creating a Relief Matrix Item*
- ◆ *Setting Matrix Items Selection*
- ◆ *Changing Matrix Items Sensitivity*

Predefined Matrix Item Classes

Below is a list of subclasses of `IlvAbstractMatrixItem`:

- ◆ `IlvLabelMatrixItem` defines a matrix item as a label.
- ◆ `IlvFilledLabelMatrixItem` defines a matrix item as a label with a filled background.
- ◆ `IlvBitmapMatrixItem` defines a matrix item as a bitmap.

- ◆ `IlvIntMatrixItem` defines a matrix item as an integer.
- ◆ `IlvFilledIntMatrixItem` defines a matrix item as an integer with a filled background.
- ◆ `IlvFloatMatrixItem` defines a matrix item as a floating-point value.
- ◆ `IlvFilledFloatMatrixItem` defines a matrix item as a floating-point value with a filled background.
- ◆ `IlvDoubleMatrixItem` defines a matrix item as a double-precision floating-point value.
- ◆ `IlvFilledDoubleMatrixItem` defines a matrix item as a double-precision floating-point value with a filled background.
- ◆ `IlvGraphicMatrixItem` defines a matrix item as a graphic object.
- ◆ `IlvGadgetMatrixItem` defines a matrix item as a gadget. This type of matrix item differs from `IlvGraphicMatrixItem` objects in that it can be active in a matrix.
- ◆ `IlvGadgetItemMatrixItem` defines a matrix item as a gadget item.

Creating a New Subclass of Matrix Items

If the predefined subclasses of the `IlvAbstractMatrixItem` class (see *Predefined Matrix Item Classes* on page 307) do not fit your needs, you can create your own matrix item subclass. This section describes how to properly register a new matrix item class. Typically, this will enable your matrix item class to be persistent.

The code sample located below is taken from the sample `edit`. This sample can be found in `ILVHOME/samples/gadgets/table/src/edit.cpp`, where `ILVHOME` is the root directory under which IBM ILOG Views has been installed.

The class described here is a subclass of the `IlvFloatMatrixItem` class. It overrides the `IlvFloatMatrixItem::getFormat` method to give an access to the display format.

However, the important point here is not the class itself, but the registration of the class through the use of macros.

```
class FormattedFloatItem : public IlvFloatMatrixItem
{
public:
    FormattedFloatItem(IlvFloat value, const IlString& format)
        : IlvFloatMatrixItem(value),
          _format(format)
    {
    }
    void setFormat(const IlString& format)
    {
        _format = format;
    }
    virtual const char* getFormat() const
    {
        return (const char*)_format;
    }
    DeclareMatrixItemInfo();
    DeclareMatrixItemIOConstructors(FormattedFloatItem);
protected:
    IlString _format;
};
```

The macro `DeclareMatrixItemInfo` declares the methods and members needed to handle class information.

The macro `DeclareMatrixItemIOConstructors` declares the i/o and copy constructors. These constructors are defined in the following way:

```
FormattedFloatItem::FormattedFloatItem(const FormattedFloatItem& source)
    : IlvFloatMatrixItem(source),
      _format(source._format)
{
}
FormattedFloatItem::FormattedFloatItem(IlvDisplay* display,
                                       IlvInputFile& is)
    : IlvFloatMatrixItem(display, is),
      _format()
{
    _format.readQuoted(is.getStream());
}
```

The `write` method is needed because the macro `DeclareMatrixItemInfo` was used in the class declaration. It simply calls the superclass `write` method, and writes the `_format`

member into the stream. To create a read-only subclass of `IlvAbstractMatrixItem`, use the macro `DeclareMatrixItemInfoRO`.

```
void
FormattedFloatItem::write(IlvOutputFile& os) const
{
    IlvFloatMatrixItem::write(os);
    os.getStream() << " ";
    _format.writeQuoted(os.getStream());
    os.getStream() << " ";
}

```

The implementation of the `copy` and `readItem` methods are defined using the following macro:

```
IlvPredefinedMatrixItemIOMembers(FormattedFloatItem);
```

Finally, the class is registered as a subclass of `IlvFloatMatrixItem`.

```
IlvRegisterMatrixItemClass(FormattedFloatItem, IlvFloatMatrixItem);
```

Adding and Removing Matrix Items

You can add an item to a matrix at a specific location with the member function `IlvMatrix::set` and remove it with `IlvMatrix::remove`. The `IlvMatrix::getItem` member function retrieves an item given its position in the matrix.

Redrawing Matrix items

After adding or removing an item, or modifying it in any way, you must call `IlvMatrix::reDrawItem` to redraw it. You can also wait until all the modifications are made and call `IlvGadget::reDraw` at the very end to redraw the entire matrix.

Aligning Matrix Items

A matrix item can be centered within a cell or be aligned with the right or left border of the cell.

Left	Center	Right

Figure 14.2 Aligning Items in a Cell

You can change the alignment of a matrix item using the member function `IlvMatrix::setItemAlignment`. You have to redraw the matrix item for the modifications to take effect. See the section *Redrawing Matrix items* on page 310.

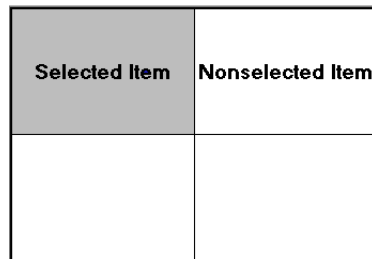
Note: `IlvGadgetMatrixItem` objects and `IlvGraphicMatrixItem` objects holding nonzoomable graphics occupy the full rectangle of the cell and cannot be aligned. For information on these classes, see *Predefined Matrix Item Classes* on page 307.

Creating a Relief Matrix Item

You can create a special relief effect for a matrix item. An item in relief has the same appearance as a button. When selected, a relief matrix item appears like a clicked button. To have a button appear in relief, use `IlvMatrix::setItemRelief`. You must call the `redrawItem` member function for the modifications to take effect. See *Redrawing Matrix items* on page 310.

Setting Matrix Items Selection

Matrix items can be selected. A selected matrix item is surrounded by a filled rectangle.



Selected Item	Nonselected Item

Figure 14.3 *Selected and Nonselected Matrix Items*

You can select a single matrix item with the member function `IlvMatrix::setItemSelected` and an entire row or column with `IlvMatrix::setColumnSelected` and `IlvMatrix::setRowSelected`. In single selection mode `setSelectedItem` does not deselect the previously selected item. See *Selection Modes* on page 312.

Once you have selected an item, you must redraw it. See *Redrawing Matrix items* on page 310.

You can change the way the selection is drawn by overriding the member function `IlvMatrix::drawSelection` in subclasses.

To retrieve the first item selected in a matrix, call `IlvMatrix::getFirstSelected`.

Changing Matrix Items Sensitivity

Matrix items can be sensitive or nonsensitive. Nonsensitive matrix items cannot be selected nor edited and appear dimmed by default as shown below. You can use the member function `IlvMatrix::setItemGrayed` with `ILFalse` as the parameter to make them look like sensitive items (that is, not grayed).

Sensitive	Nonsensitive

Figure 14.4 Sensitive and Nonsensitive Matrix Items

To change the sensitivity of an item, use the member function `IlvMatrix::setItemSensitive`. You must redraw the item for the modifications to take effect. See *Redrawing Matrix items* on page 310.

Note: If the matrix item is a graphic object or a gadget, modifying the sensitivity does not affect the drawing of the item.

You can also set a matrix item as read only with `IlvMatrix::setItemReadOnly`. Read-only matrix items cannot be edited but can be selected.

Handling Events

This section describes the standard matrix behavior implemented by the member function `IlvMatrix::handleMatrixEvent`. The following topics are covered:

- ◆ *Selection Modes*
- ◆ *Editing a Matrix Item*
- ◆ *Item Callback*
- ◆ *Activate Callback*
- ◆ *Using Gadgets in a Matrix*
- ◆ *Modifying `handleEventMatrix`*

Selection Modes

There are two selection modes for matrices: single (or exclusive) selection and multiple selection. To set the selection mode, use `IlvMatrix::setExclusive` with `ILTrue` as its

parameter if you want to specify the single selection mode or `IlvFalse` to specify the multiple selection mode.

In single selection mode, only one item can be selected at a time. This mode has two submodes:

- ◆ Single selection—In this mode, you can select only one item at a time. When the selection changes, the previous selected item is deselected.
- ◆ Single browse—This mode is similar to the previous one except that clicking the selected item with the middle mouse button cancels the selection.

In multiple selection mode, several items can be selected at the same time. This mode has two submodes:

- ◆ Multiple browse—In this mode, you can select several items at the same time either by clicking them or dragging the mouse. Similarly, you can deselect several items by clicking them or by dragging the mouse with the middle button.
- ◆ Extended—In this mode, you can select several items at the same time either by clicking them or by dragging the mouse. You can extend the selection by pressing the Shift and CTRL keys while selecting items. You can also specify the direction of the extended selection by using `IlvMatrix::setExtendedSelectionOrientation`

To specify the selection submode, use `IlvMatrix::setBrowseMode`.

When the user selects a matrix item or cancels the selection, the `Main` callback of the matrix is called.

Note: *You cannot select items that are not sensitive.*

Editing a Matrix Item

If editing is allowed for the matrix (see `IlvMatrix::allowEdit`), you can edit matrix items. Nonsensitive or read-only matrix items cannot be edited. When an item is being edited, an editor is displayed over it as shown in the illustration below. The base class for matrix item editors is the `IlvMatrixItemEditor` class. It encapsulates an `IlvGraphic` object that will be used to display and edit the matrix item. The default editor class used by an `IlvMatrix` is the `IlvDefaultMatrixItemEditor` class. It uses an `IlvTextField` object to edit matrix items. You can change this behavior by using the `IlvMatrixItemEditorFactory` class.

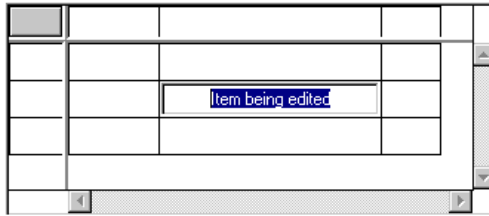


Figure 14.5 Editing a Matrix Item

To edit an item, select it first with the mouse and click it, or press the F2 key to edit the last selected item. To validate your modifications, press the Enter key in the text field or click in another cell. The virtual member function `IlvMatrix::validate` is called. Its default implementation invokes the callback associated with the item, if any. See “Item Callback” below. If there is none, it invokes the secondary callback of the matrix. See *Associating a Callback with a Gadget* on page 210.

To edit an item by code, call `IlvMatrix::editItem`.

`IlvMatrix::getEditedItem` returns the location of the matrix item being edited.

Item Callback

You can attach a callback to each matrix item. When the user validates the editing of an item, its associated callback is invoked. This callback is defined by:

```
typedef void (*IlvMatrixItemCallback)(IlvMatrix* matrix,
                                     IlUShort column,
                                     IlUShort row,
                                     IlvAny arg);
```

where `matrix` specifies the matrix that contains the item, `column` and `row` the location of the item that invoked the callback, and `arg` an argument passed when installing the callback.

To attach a callback to an item, use `IlvMatrix::setItemCallback`.

Activate Callback

When the user double-clicks an item or presses the Enter key, the member function `IlvMatrix::activateMatrixItem` is called. By default, this method invokes the Active Item callback. To set this callback, use

`IlvMatrix::ActivateMatrixItemCallbackType`. See *Associating a Callback with a Gadget* on page 210.

Note: By default, when the user double-clicks a matrix item, this item is ready for editing. In this case, the member function `IlvMatrix::activateMatrixItem` is not called. If you want to override this default behavior, call `IlvMatrix::allowEditOnDoubleClick` with `IlFalseIlFalse` as the parameter.

Using Gadgets in a Matrix

Gadget matrix items do not have the same behavior as other items. They react to events by using their `handleEvent` member function. This means that a gadget inside a matrix behaves like a gadget outside a matrix. The matrix defines a specific gadget matrix item that can have the keyboard focus. You can specify this item with `IlvMatrix::setFocus`.

When navigating through the matrix using the arrow keys, you can reach a cell that contains a gadget matrix item. You may want to either continue navigating, or send events to the gadget matrix item. To continue navigating, use the arrow keys to leave the cell. To send events to the gadget matrix item, you can either press a key that the gadget will catch (any key except the arrow keys) or press `CTRL+Enter`. The gadget matrix item will receive all the keyboard inputs until it receives another `CTRL+Enter`, or a key that it does not handle.

Modifying `handleEventMatrix`

You may need to modify the default behavior of a matrix in a subclass of `IlvMatrix` by redefining the method `IlvMatrix::handleMatrixEvent`. Some methods can help you as those shown below.

This method returns the column and row to which the mouse points:

```
virtual IlvBoolean  pointToPosition(IlvPoint& p,
                                   IlUShort& c,
                                   IlUShort& r,
                                   const IlvTransformer* t = 0) const;
```

This method returns the matrix item to which the mouse points:

```
virtual IlvAbstractMatrixItem* pointToItem(IlvPoint& p,
                                             IlUShort& c,
                                             IlUShort& r,
                                             const IlvTransformer* t = 0) const;
```

Using Gadget Items in a Matrix

A matrix can hold gadget items via the class `IlvGadgetItemMatrixItem`, a subclass of `IlvAbstractMatrixItem`. Instances of this class encapsulate a gadget item and therefore benefit from all its features. See Chapter 12, *Gadget Items*.

Picture and Label Visibility

You can specify whether all the pictures in a matrix should be displayed by calling the method `IlvMatrix::showPicture`. Likewise, you can use `IlvMatrix::showLabel` to define the visibility of all its labels. By default, the matrix displays both labels and pictures.

Note: You can override these global settings for a specific item by using the API of the `IlvGadgetItem` class. For details, see the methods `IlvGadgetItem::showLabel` and `IlvGadgetItem::showPicture`.

Label and Picture Position

You may want to change the position of the item labels relative to their pictures. To do so, use the method `IlvMatrix::setLabelPosition`. By default, the label is placed to the right of the picture (`IlvRight`).

Note: You can override this global setting for a specific item by using the API of the `IlvGadgetItem` class. For details, see the method `IlvGadgetItem::setLabelPosition`.

Editing Items

You can edit gadget items located in a matrix. See *Editing Gadget Items* on page 285.

Dragging and Dropping Items

The `IlvMatrix` class provides an easy-to-use drag-and-drop mechanism. See *Dragging and Dropping Gadget Items* on page 286.

Tooltips

Matrices can display tooltips when the mouse pointer is over partially visible items.

Using IlvSheet

A sheet is a particular type of matrix implemented by the class `IlvSheet`. See *Using IlvMatrix* on page 305. In a sheet, fixed rows and columns are delimited by a relief line.

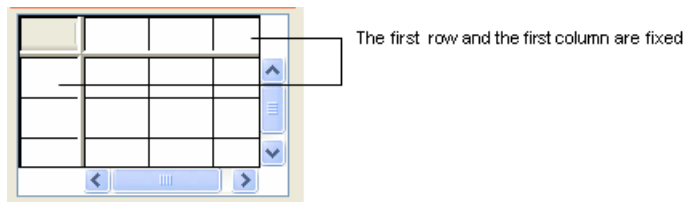


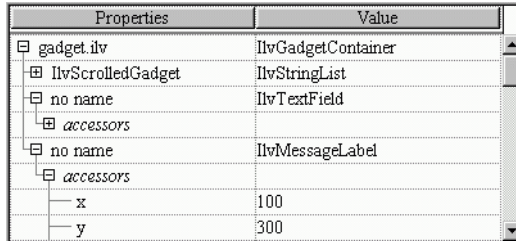
Figure 14.6 A Sheet

`IlvSheet` has all the behavior of the class `IlvMatrix`. In addition, it allows the user to dynamically resize the columns or rows. This can be done in two ways:

- ◆ By clicking in the fixed columns or rows on the grid line and dragging to resize the column or row.
- ◆ By double-clicking in the fixed columns or rows on the grid line to give the column or row the size of its larger item.

Using IlvHierarchicalSheet

The `IlvHierarchicalSheet` class is a subclass of `IlvSheet` that displays a tree structure in one of its columns. It can be considered as a special `IlvTreeGadget` object that handles several columns. The tree items are of the type `IlvTreeGadgetItem`, which means that the API used to handle a tree hierarchy is very close to the `IlvTreeGadget` object. See *Using IlvTreeGadget* on page 271.



Properties	Value
gadget.ilv	IlvGadgetContainer
IlvScrolledGadget	IlvStringList
no name	IlvTextField
accessors	
no name	IlvMessageLabel
accessors	
x	100
y	300

Figure 14.7 A Hierarchical Sheet

This section covers the following topics:

- ◆ *Changing the Tree Hierarchy*
- ◆ *Navigating through a Tree Hierarchy*
- ◆ *Changing the Characteristic of a Tree Item*
- ◆ *Expanding and Collapsing a Gadget Item*
- ◆ *Changing the Look of the Tree Gadget Hierarchy*
- ◆ *Event Handling and Callbacks*

Changing the Tree Hierarchy

The hierarchical sheet has an invisible root item that can be retrieved using the `IlvHierarchicalSheet::getRoot` member function.

Changing a Hierarchy

When you want to modify the tree hierarchy, you must not use `IlvSheet` member functions, such as `set`, `removeRow`, and so on. Instead, use the `IlvHierarchicalSheet` methods described below.

To create a hierarchical list of items, you can do the following:

- ◆ Create tree gadget items as explained in *Creating a Hierarchy* on page 272 and add them one by one to the hierarchical sheet with `IlvHierarchicalSheet::addItem` member function.

- ◆ Create a complete new hierarchy and add it to the tree gadget in a single operation. To do so, create tree gadget items as explained above and add them as children using `IlvTreeGadgetItem::insertChild`. This solution is more efficient than the first one.

Then you can add the root of your new hierarchy to the tree with `IlvHierarchicalSheet::addItem` as shown below:

```
IlvTreeGadgetItem* item = new IlvTreeGadgetItem("New Item");
item->insertChild(new IlvTreeGadgetItem("Leaf 1"));
item->insertChild(new IlvTreeGadgetItem("Leaf2"));
hsheet->addItem(0 /* hsheet->getRoot() */, item);
```

Removing Items

When you remove an item from the hierarchical sheet, all its children are also removed from the tree. Use `IlvHierarchicalSheet::removeItem` to remove a single item from a tree or `IlvHierarchicalSheet::removeAllItems` to remove all its items at once.

Note: When you add a new item to the tree gadget, its corresponding row is created automatically. Similarly, when you remove an item, its row is deleted.

Navigating through a Tree Hierarchy

To move inside a hierarchical tree, use the member functions described in *Navigating Through a Tree Hierarchy* on page 273. You can also use

`IlvHierarchicalTree::getTreeItem` and `IlvHierarchicalSheet::getItemRow`.

Changing the Characteristic of a Tree Item

See *Changing the Characteristic of an Item* on page 273.

Expanding and Collapsing a Gadget Item

You can expand or collapse a gadget item by clicking its Expand button. Expanding an item shows all its subitems; collapsing an item hides all its subitems. You can also perform the same operations using `IlvHierarchicalSheet::shrinkItem` and `IlvHierarchicalSheet::expandItem`.

When an item becomes invisible because one of its parents has been collapsed, its corresponding row in the sheet disappears. Note, however, that it is not deleted.

Changing the Look of the Tree Gadget Hierarchy

The lines that link items to their parents can be displayed or hidden using the `IlvHierarchicalSheet::showLines` member function.

You can define the indentation between an item and its parent using the `IlvHierarchicalSheet::setIndent` member function.

Event Handling and Callbacks

The Expand Callback

When the user expands a tree gadget item, the Expand callback is invoked. The callback type can be retrieved with `IlvHierarchicalSheet::ExpandCallbackType`. See *Associating a Callback with a Gadget* on page 210.

The Shrink Callback

When the user collapses a tree gadget item, the Shrink callback is invoked. The callback type can be retrieved with `IlvHierarchicalSheet::ShrinkCallbackType`. See *Associating a Callback with a Gadget* on page 210.

Panes

You can group the various elements that make up a graphical user interface, such as graphic panels, tool bars, and menu bars, inside panes of various sizes to create highly intuitive and customizable applications.

This chapter explains what panes are and how to use them in your graphical applications. It covers the following topics:

- ◆ *Introducing Panes*
- ◆ *Creating Panes*
- ◆ *Adding Panes to Paned Containers*
- ◆ *Resizing Panes*

Introducing Panes

The IBM® ILOG® Views Gadgets library supports panes. A pane is a graphical area that displays any kind of drawing, such as `IlvGraphic` or `IlvView` objects.

Panes are objects of the class `IlvPane` that are stored in paned containers of the class `IlvPanedContainer`. A paned container can be either vertical or horizontal. In vertical paned containers, panes are arranged from top to bottom, whereas in horizontal paned containers they are arranged from left to right. Panes inside a vertical paned container have

all the same width, but their height can vary. Similarly, panes inside a horizontal paned container have all the same height, but their width can vary.

A pane can encapsulate a paned container, allowing you to build complex, nested pane structures, as illustrated in the following figure.

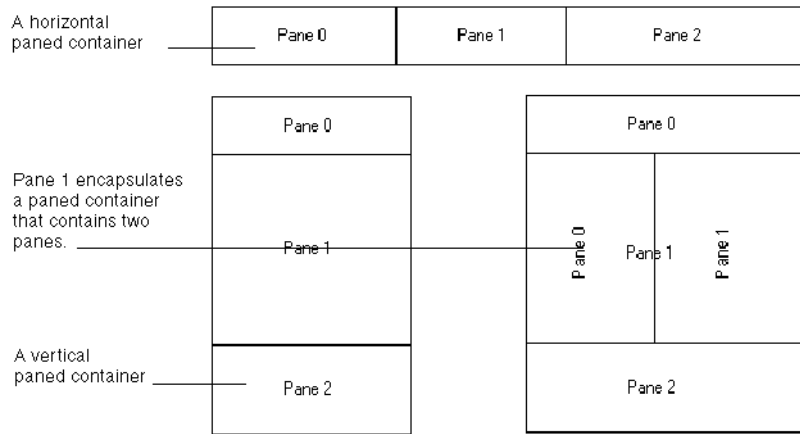


Figure 15.1 *Horizontal and Vertical Paned Container and Encapsulated Paned Container*

The following figure shows an application main window that implements panes:

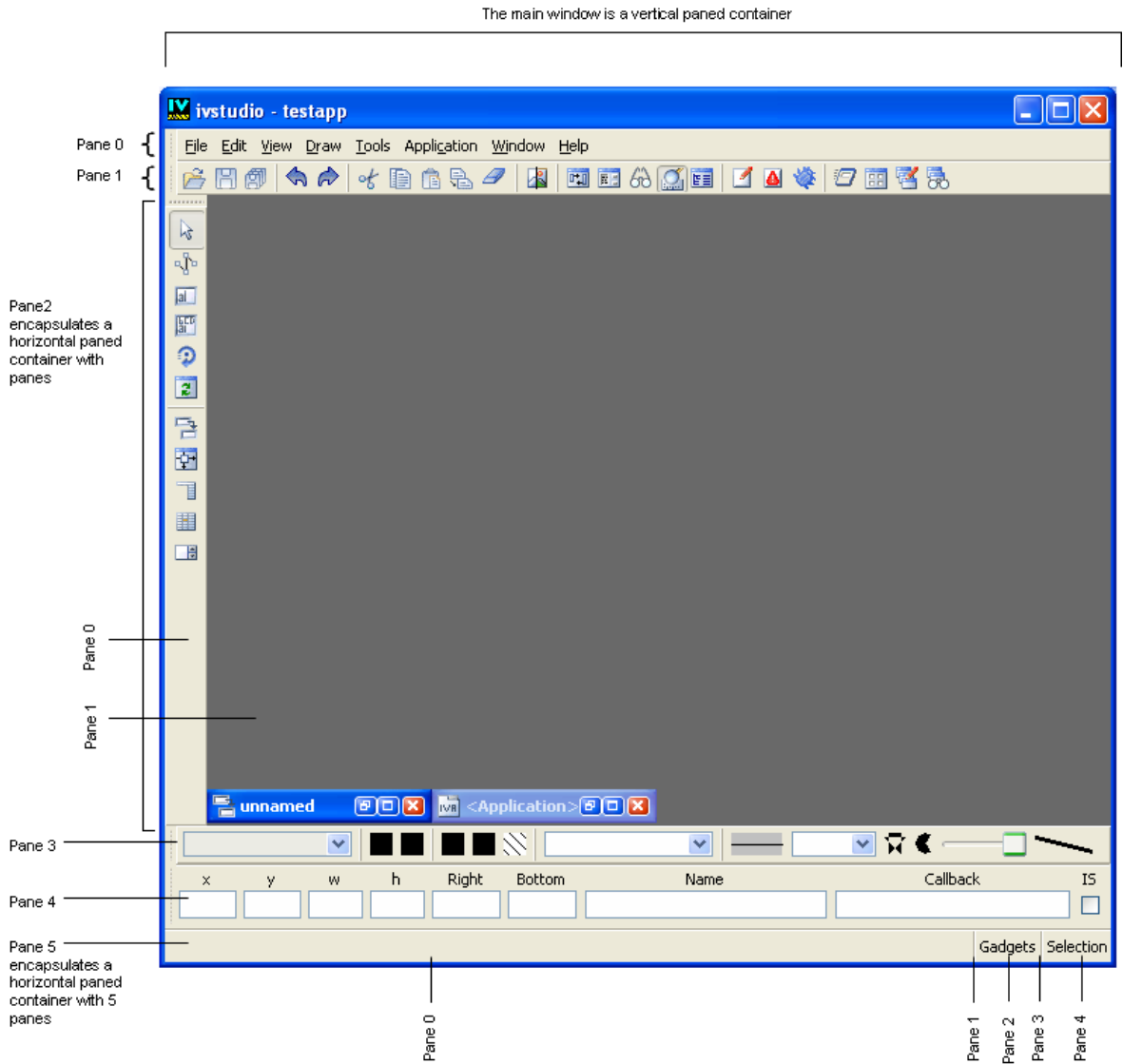


Figure 15.2 Application Main Window Made Up of Several Panes

Creating Panes

A pane is an instance of the `IlvPane` class. Since this class is abstract, you must subclass it or use one of its predefined subclasses:

- ◆ `IlvViewPane`, which encapsulates any `IlvView` object.
- ◆ `IlvGraphicPane`, which encapsulates any `IlvGraphic` object.

Most of the time, you do not have to subclass the `IlvPane` class as its predefined subclasses are appropriate for nearly all application needs.

This section discusses the following:

- ◆ *Creating a Graphic Pane*
- ◆ *Creating a View Pane*
- ◆ *Showing or Hiding a Pane*

Creating a Graphic Pane

The following example demonstrates how to create a graphic pane (`IlvGraphicPane`).

First we create the `IlvGraphic` object we want to add to the graphic pane. Here, we encapsulate an `IlvTreeGadget` object.

```
IlvDisplay* display = ...
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
```

Then we create a graphic pane:

```
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree", tree);
```

The first argument provided to the constructor is a string representing the name of the pane.

Creating a View Pane

The following example shows how to create a view pane (`IlvViewPane`).

First we create the `IlvView` object we want to add to the view pane:

```
IlvView* view = new IlvView(parent, IlvRect(0, 0, 100, 100));
```

Then we create the view pane:

```
IlvViewPane* viewPane = new IlvViewPane("View", view);
```

The first argument provided to the constructor is a string representing the name of the pane.

Note: *The view used in the view pane must be a subview. That is why the encapsulated view was created with the `IlvView` constructor that takes a parent view as its first argument.*

Showing or Hiding a Pane

You can show or hide a pane using the `IlvPane::show` and `IlvPane::hide` member functions. A hidden pane does not appear in its paned container.

Note: *If you modify the layout of a paned container by adding or removing panes or by showing or hiding panes, you must call the `IlvPanedContainer::updatePanes` member function for your changes to become effective.*

Adding Panes to Paned Containers

A paned container is an instance of the `IlvPanedContainer` class, a subclass of `IlvGadgetContainer`, to which panes must be added.

This section covers the following topics:

- ◆ *Creating a Paned Container*
- ◆ *Modifying the Layout of a Paned Container*
- ◆ *Retrieving Panes*
- ◆ *Encapsulating a Paned Container in a View Pane*

Creating a Paned Container

When creating a paned container, you must specify its direction (horizontal or vertical). In a vertical paned container, panes are arranged from top to bottom. In a horizontal pane, they are arranged from left to right.

The following code sample creates a vertical paned container as a top view:

```
IlvPanedContainer* container = new IlvPanedContainer(display,
                                                    "Paned Container",
                                                    "Paned Container",
                                                    IlvRect(0, 0, 500, 500),
                                                    IlvVertical);
```

You can retrieve the specified orientation and modify it using the member functions `IlvPanedContainer::getDirection` and `IlvPanedContainer::setDirection`.

Once you have created a paned container, you can add panes to it with the member function `IlvPanedContainer::addPane` or remove panes from it with `IlvPanedContainer::removePane`.

Modifying the Layout of a Paned Container

If you modify the current layout of a paned container by adding, removing, showing, or hiding panes, you must call the `IlvPanedContainer::updatePanes` member function to make your changes effective.

```
container->addPane (pane1) ;
container->addPane (pane2) ;
container->addPane (pane3) ;
container->updatePanes () ;
```

Retrieving Panes

You can use the member function `IlvPanedContainer::getCardinal` to know the number of panes that a given paned container handles.

The `IlvPanedContainer::getPane` member functions lets you retrieve a pane using its index or using its name.

You can get the index of a specific pane with the `IlvPanedContainer::getIndex` member function.

***Note:** Paned containers reference the panes they hold using indexes. However, we strongly recommend that you do not reference panes using their indexes, because these can change for internal reasons. Instead, use the member function `IlvPane::setName` to identify panes.*

Encapsulating a Paned Container in a View Pane

Because the class `IlvPanedContainer` inherits from `IlvGadgetContainer`, itself a subclass of `IlvView`, you can encapsulate a paned container inside a view pane.

Encapsulating a paned container in a view pane allows you to build complex nested pane structures, as shown in Figure 15.1 on page 321.

The following code sample encapsulates a horizontal paned container in a view pane.

First we create the main vertical paned container:

```
IlvPanedContainer* container = new IlvPanedContainer (display,
                                                    "Paned Container",
                                                    "Paned Container",
                                                    IlvRect (0, 0, 500, 500),
                                                    IlvVertical);
```

Then we create a horizontal paned container and encapsulate it in a view pane:

```
IlvPanedContainer* innerContainer = new IlvPanedContainer(container,
                                                    IlvRect(0, 0, 500,200),
                                                    IlvHorizontal);
IlvViewPane* viewPane = new IlvViewPane("ViewPane", innerContainer);
```

Note: In our example, we have created `innerContainer` as a subview of `container`. Although this practice is not mandatory, we strongly recommend that you proceed that way when creating your own applications. If you do not specify `container` as the parent of `innerContainer`, it will be reparented when added to `container`.

The last step consists of adding the view pane to the main paned container:

```
container->addPane(viewPane);
```

Note: You can get the view pane that encapsulates a given paned container using the `IlvPanedContainer::getViewPane` member function. If no view pane encapsulates the paned container, this member function returns 0.

Resizing Panes

Panes can be resized.

This section covers the following topics:

- ◆ *Setting the Resize Mode and the Minimum Size of a Pane*
- ◆ *Resizing Panes With Sliders*

Setting the Resize Mode and the Minimum Size of a Pane

When you resize a paned container, the panes it holds are resized according to their resizing mode. A pane can have one of three resizing modes:

- ◆ **Fixed**—Fixed panes are never resized.
- ◆ **Elastic**—Elastic panes are always resized.
- ◆ **Resizable**—Resizable panes are resized only if their paned container does not include elastic panes.

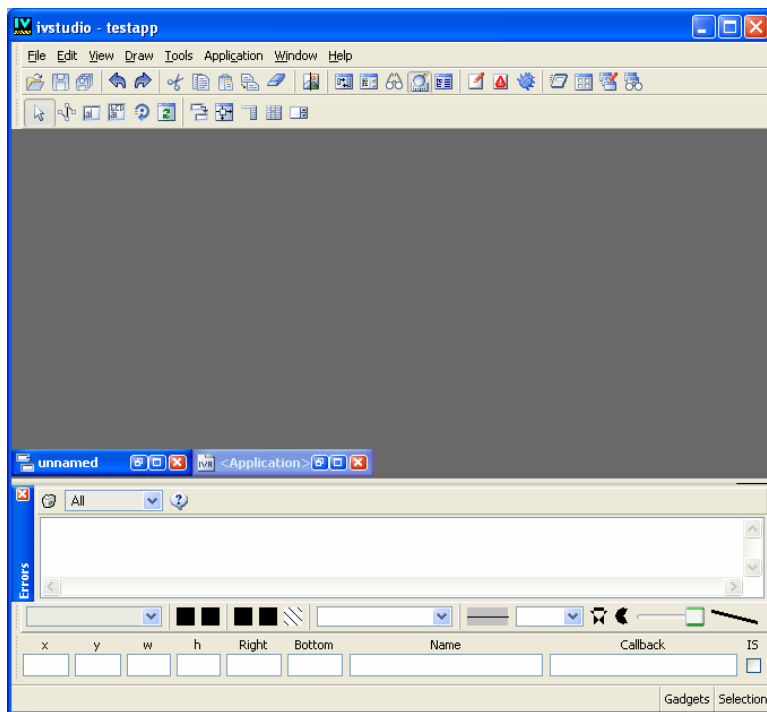
To set the resize mode of a pane, use the `IlvPane::setResizeMode` member function. By default, the resize mode of new panes is fixed.

You can also provide a minimum size for a pane. To set a minimum size for a pane, use the `IlvPane::setMinimumSize` member function. The minimum size of new panes is 1 by default. You cannot make a pane smaller than the specified minimum size.

Note: *The resize mode and the minimum size of a pane can be defined for both the horizontal and vertical directions.*

Resizing Panes With Sliders

A slider pane is an instance of the class `IlvSliderPane`, a subclass of `IlvGraphicPane`, which you can drag to resize adjacent panes.



Click on the slider panel and drag it up or down to resize adjacent panes. Panes are resized when you release the mouse button.

Figure 15.3 *Slider Pane*

Using Automatic Slider Creation

By default, a paned container automatically creates slider panes between resizable and elastic panes.

Note: *Slider panes are actually created only after you call the member function `IlvPanedContainer::updatePanes`. This is very important as it affects the index number originally assigned to the panes. For example, if you create an empty container to which you add two resizable panes, their indexes will be 0 and 1, respectively. After calling `updatePanes`, the indexes of the resizable panes will be 0 and 2, the slider pane being assigned the index number 1. See [Retrieving Panes](#) on page 325.*

When the paned container creates automatic slider panes, it calls the `IlvPanedContainer::createSliderPane` member function, which you can override to create custom slider panes.

If you do not want that slider panes be created automatically, you can call the `IlvPanedContainer::manageSliders` member function with `false` as its argument. If you disable this feature, and if you still want resizable and elastic panes to be resizable using a slider, you must create sliders panes by hand and add them to the paned container.

Docking Panes and Containers

The IBM® ILOG® Views Gadgets library supports docking panes.

This section explains what docking panes are and how to use them in your graphical applications. Before reading this section, be sure that you are familiar with panes. Panes are discussed in detail in Chapter 15, *Panes*.

This chapter covers the following topics:

- ◆ *Introducing Docking Panes and Dockable Containers*
- ◆ *Creating Docking Panes*
- ◆ *Controlling Docking Operations*
- ◆ *Using Docking Bars*
- ◆ *Building a Standard Application With Docking Panes*

Samples

See the ViewFile Application tutorial.

Introducing Docking Panes and Dockable Containers

A dockable container is a particular type of paned container to which you can dock panes and undock them. Docking a pane means adding it to a dockable container at a given location interactively. Undocking a pane means removing it from its dockable container to put it inside a special top view interactively.

The classes in the IBM® ILOG® Views Gadgets library that implement docking panes are `IlvDockable` and `IlvDockableContainer`.

In the following illustration, all the panes that make up the graphical interface are docked.

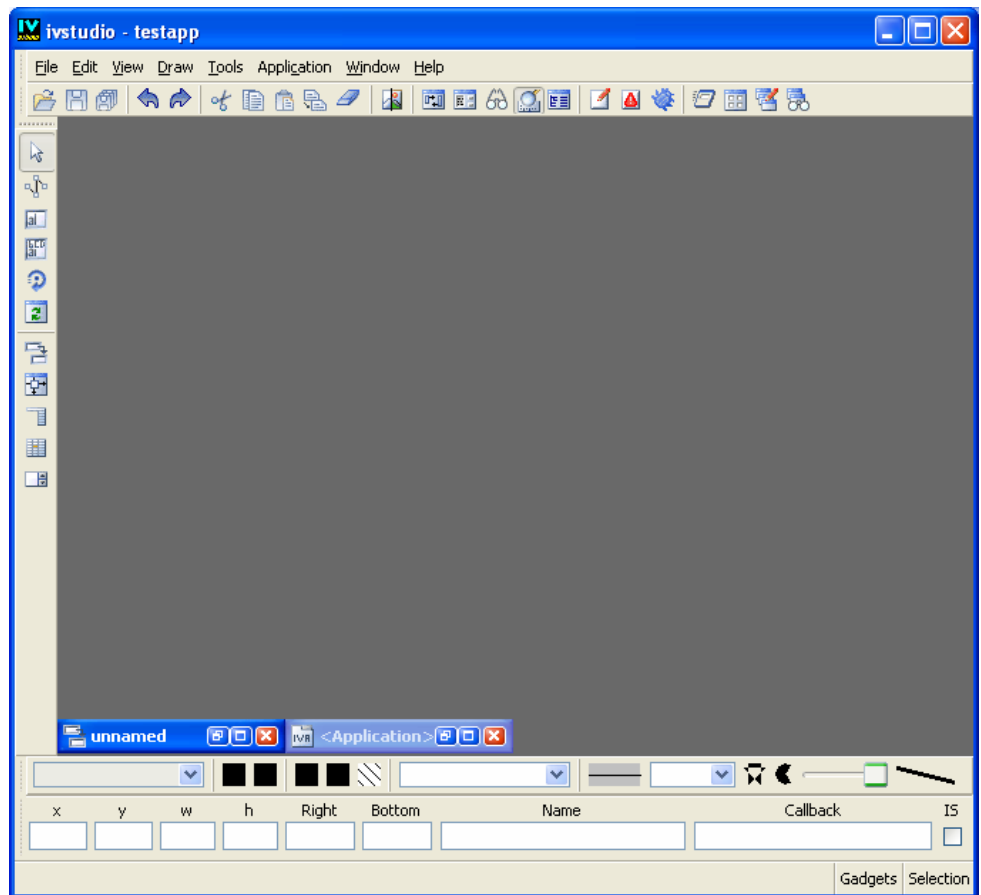


Figure 16.1 GUI with Docked Panes

In the following illustration, the main menu bar has been undocked and floats inside a top window.

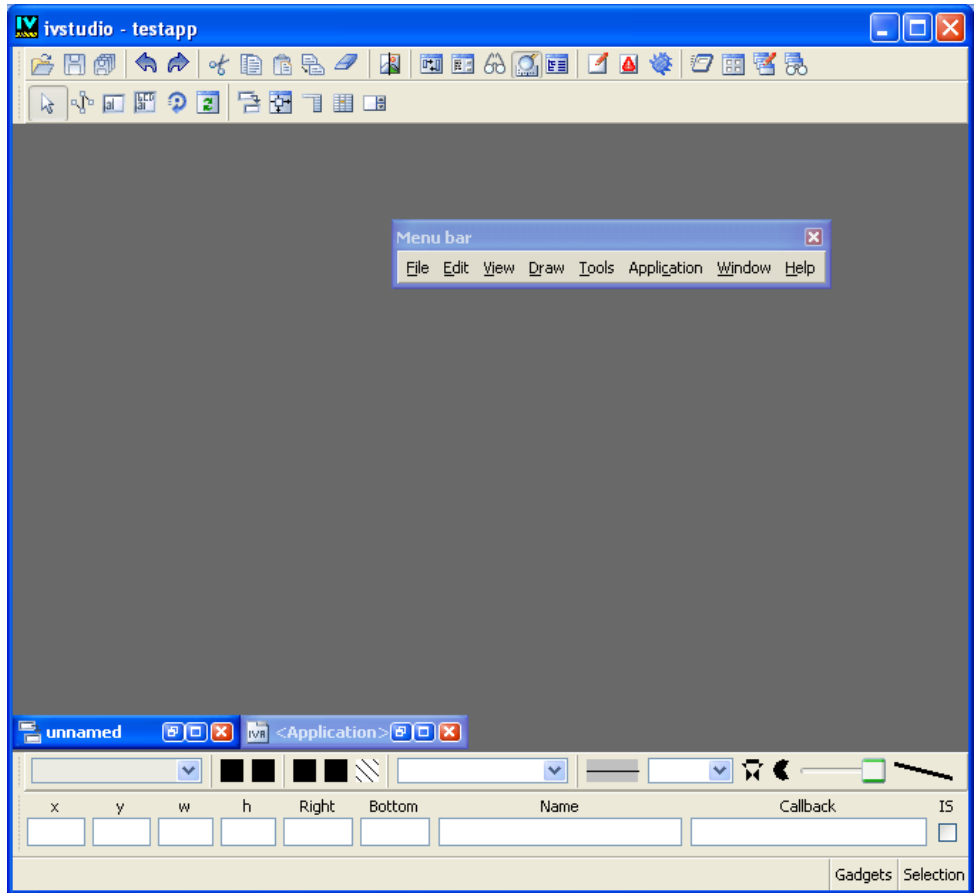


Figure 16.2 Main Menu Bar Undocked

In the following illustration, the main menu bar is docked again to the right side of the GUI main window.

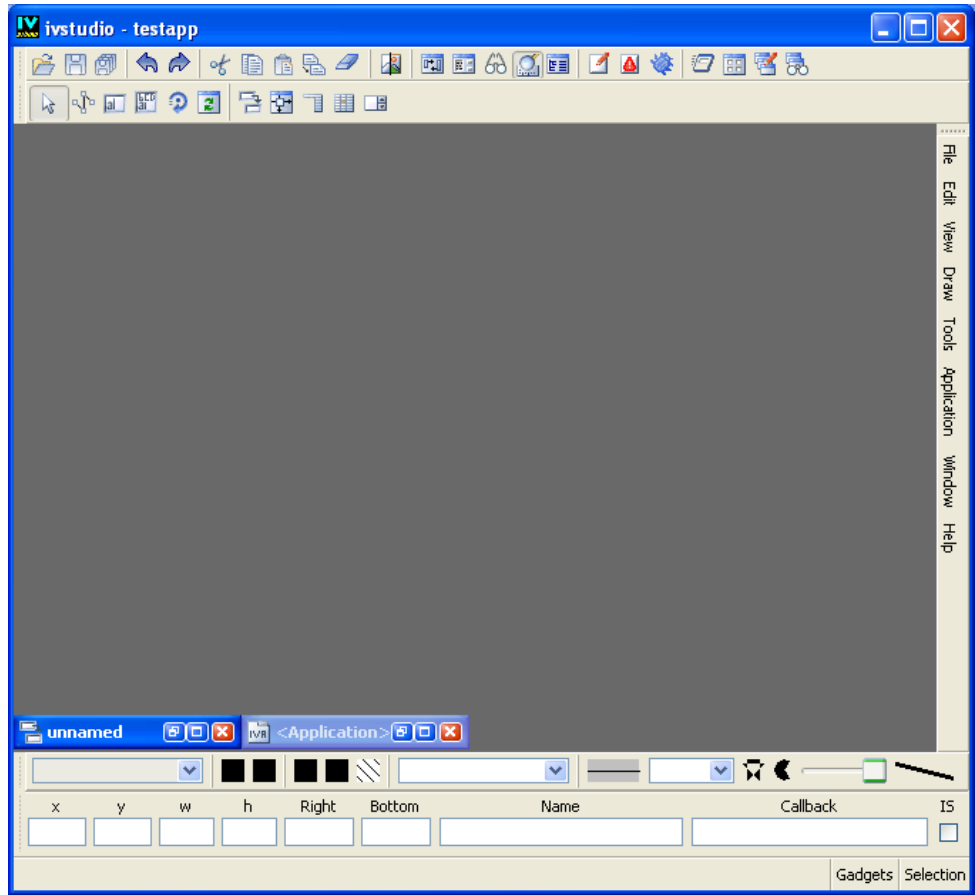


Figure 16.3 Main Menu Bar Redocked

Creating Docking Panes

Creating a docking pane is very much the same as creating a normal pane as illustrated in the following two code samples. Panes are described in Chapter 15, *Panes*.

Below, the “Tree” pane is added to a paned container with the `addPane` member function:

```

IlvPanedContainer* container = new IlvPanedContainer(display,
                                                    "Paned Container",
                                                    "Paned Container",
                                                    IlvRect(0, 0, 500, 500),
                                                    IlvVertical);
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree", tree);

```

```
container->addPane(graphicPane);
```

Below, the same “Tree” pane is added to a dockable container with the member function `IlvDockableContainer::addDockingPane`, which makes it a dockable pane:

```
IlvDockableContainer* container =
    new IlvDockableContainer(display,
                            "Dockable Container",
                            "Dockable Container",
                            IlvRect(0, 0, 500, 500),
                            IlvVertical);
IlvTreeGadget* tree = new IlvTreeGadget(display, IlvRect(0, 0, 100, 100));
IlvGraphicPane* graphicPane = new IlvGraphicPane("Tree", tree);
container->addDockingPane(graphicPane);
```

In the second code sample, the paned container is of type `IlvDockableContainer`, a subclass of `IlvPanedContainer`, and panes are added to it with the member function `IlvDockableContainer::addDockingPane` to create docking panes.

Panes added to a dockable container with the `addDockingPane` member function are connected to an instance of the `IlvDockable` class, which handles docking operations for them. For more information on this class, see *Controlling Docking Operations* on page 338.

Note: If you want to use a subclass of `IlvDockable`, you should be aware that you have to connect it explicitly before calling `addDockingPane`. See *Connecting an Instance of the `IlvDockable` Class to a Pane* on page 338.

Docking panes are equipped with a handle, which you can click and drag to undock the pane. See the following illustration.

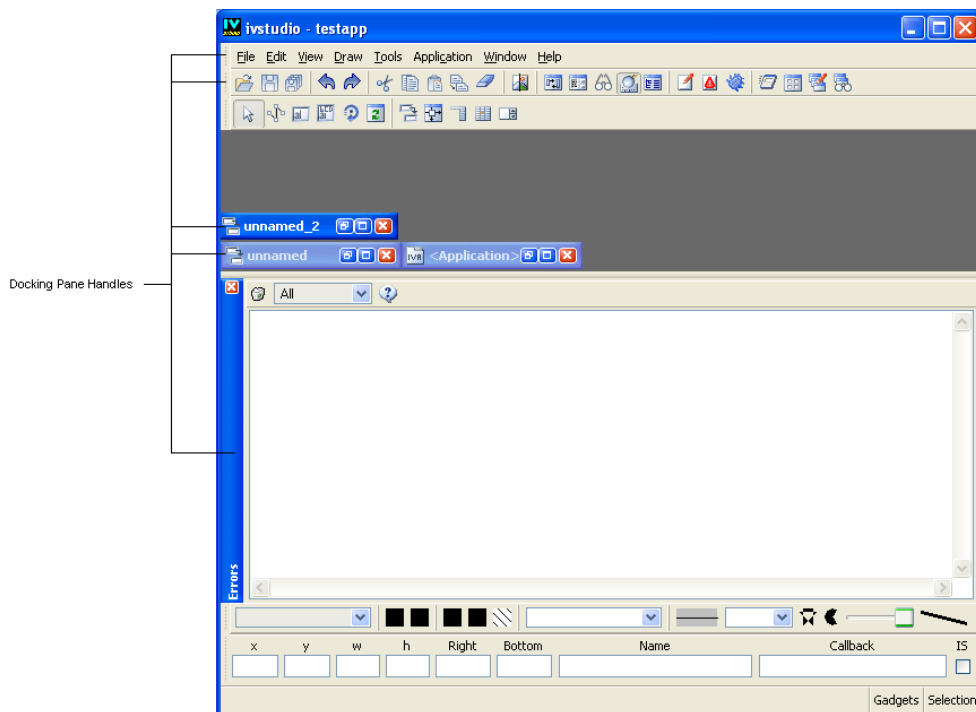


Figure 16.4 Docking Pane Handles

When you drag the pane to move it, a ghost image appears that helps you position it to its new location.

Note: Because the dockable container adds a handle to a docking pane, its index is no longer the one specified when calling the member function `IlvDockableContainer::addDockingPane`. The handle is added to the left of the pane if the target paned container is horizontal and to the top of the pane if the container is vertical. For information about pane indexing, see *Using Automatic Slider Creation* on page 328 and *Creating Orthogonal Dockable Containers* on page 335.

Creating Orthogonal Dockable Containers

Orthogonal dockable containers are an advanced feature. Use this feature if you want to create docking panes having a nonstandard behavior. For further information, see *Using the IlvDockableMainWindow Class* on page 345 where the `IlvDockableMainWindow` class that implements this feature is described.

The `IlvDockableContainer` class provides the member function `createOrthogonalDockableContainer`, which when set to `true`, modifies the behavior of the `addDockingPane` member function as follows:

- ◆ Creates an internal dockable container which is orthogonal to the main dockable container.

The “create orthogonal dockable container” feature does not apply to this internal container.

- ◆ Encapsulates the internal dockable container into a view pane.
- ◆ Adds the view pane to the dockable container.
- ◆ Adds the docking pane and its handle to the internal dockable container.

If you add a pane to a vertical dockable container when the “create orthogonal dockable container” feature is disabled, you obtain the following:

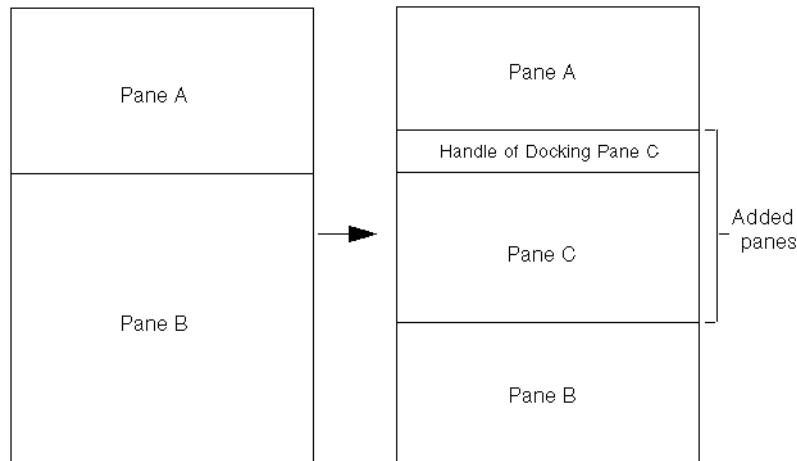


Figure 16.5 *Create Orthogonal Dockable Container Feature Disabled*

If the “create orthogonal dockable container” feature is enabled, you obtain the following:

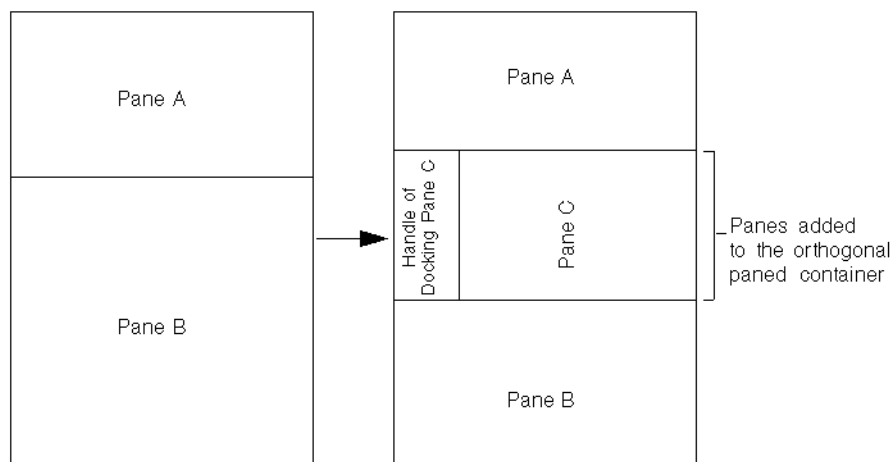


Figure 16.6 Create Orthogonal Dockable Container Feature Enabled

This feature makes it possible to dock other panes to the dockable container of Pane C to get a pane structure similar to the one shown below:

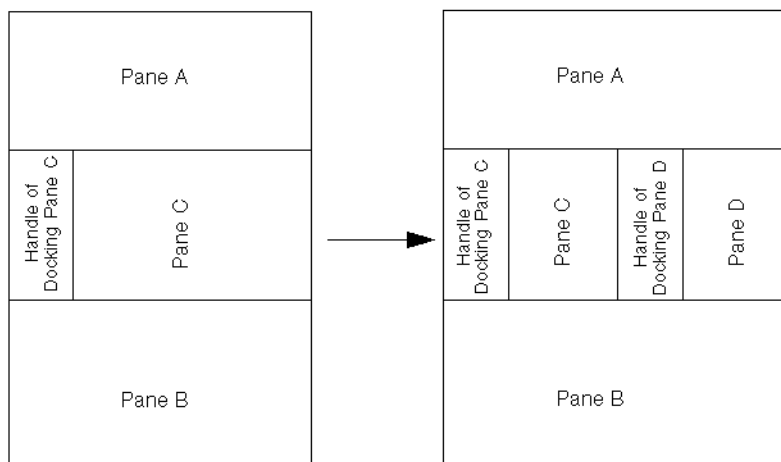


Figure 16.7 Pane D is Docked into the Dockable Container of Pane C

When the “create orthogonal dockable container” feature is enabled, the dockable container to which the pane is actually added is not the one for which you called the member function `addDockingPane`. Also, the index of the added pane might have changed. Therefore, we recommend that you retrieve the pane using its name instead of its index inside its container.

Controlling Docking Operations

You can manage docking operations relative to a specific pane with the `IlvDockable` class. Each docking pane has an instance of the `IlvDockable` class connected to it. This instance can be automatically created when adding the pane using the member function `IlvDockableContainer::addDockingPane`, or can be specified by the user.

This section covers the following topic:

- ◆ *Connecting an Instance of the `IlvDockable` Class to a Pane*
- ◆ *Docking and Undocking a Pane*
- ◆ *Filtering Docking Operations*

Connecting an Instance of the `IlvDockable` Class to a Pane

To connect an instance of the `IlvDockable` class to a pane, you first have to create an instance of the `IlvDockable` class or of a subclass like this:

```
IlvDockable* dockable = new IlvDockable();
```

Then you can set it to your pane using the static member function `IlvDockable::SetDockable`.

To retrieve the `IlvDockable` instance connected to a pane, call:

```
IlvDockable* dockable = IlvDockable::GetDockable(pane);
```

This member function returns 0 if `pane` is not a docking pane.

To retrieve the pane connected to an `IlvDockable` instance, call:

```
IlvPane* pane = dockable->getPane();
```

Docking and Undocking a Pane

When a pane is docked, you can undock it using the member function `IlvDockable::undock`.

When a pane is undocked, you can dock it using the member function `IlvDockable::dock`. This member function calls `IlvDockableContainer::addDockingPane` to dock the pane.

To know whether a pane is docked, use the `IlvDockable::isDocked` member function.

Controlling User Interactions

You can dock or undock a docking pane by double-clicking its handle.

To prevent a pane from being docked, you can press the Ctrl key while dragging it onto a dockable container.

To cancel a docking operation, press the Escape key.

Filtering Docking Operations

Potentially, a docking pane can be attached to any dockable container in your application. You can, however, control docking operations and prevent docking panes from being attached to a given container.

When you drag a pane onto a dockable container, the virtual member function `IlvDockable::acceptDocking` is called. If it returns `true`, the pane can be docked; otherwise, the operation is not allowed.

Here is a brief description of what `acceptDocking` checks:

- ◆ If the target container is the same as the current paned container, `acceptDocking` returns `IlTrue`.
- ◆ The target container is asked whether docking is allowed for the pane with the member function `IlvDockableContainer::acceptDocking`. If the dockable container returns `IlFalse`, docking is not allowed and `acceptDocking` returns `IlFalse`. By default, the member function `acceptDocking` returns the dockable state of the container. You can change this state with the member function `IlvDockableContainer::setDockable`.
- ◆ The docking direction set is compared with the direction of the target container. If both directions do not match, docking is not carried out and `acceptDocking` returns `IlFalse`. You can set the docking direction using the member function `IlvDockable::setDockingDirection`. This function is useful if you want to force a pane to always dock horizontally for example. By default, a pane can be docked both to a vertical and a horizontal container.

Using Docking Bars

Most GUI applications include docking bars. Their behavior is slightly different from that of standard docking panes.

This section introduces you to docking bars. It covers the following topics:

- ◆ *Using the `IlvAbstractBarPane` Class*
- ◆ *Customizing Docking Bars*

Using the IlvAbstractBarPane Class

The class `IlvAbstractBarPane` defines a pane specifically designed for handling toolbars and menu bars. This class is a subclass of the `IlvGraphicPane` class which encapsulates an `IlvAbstractBar` object. It is responsible for managing the bar orientation.

When a docking bar is docked, its direction must change according to its new location.

The following illustrations show the same toolbar oriented horizontally and vertically.

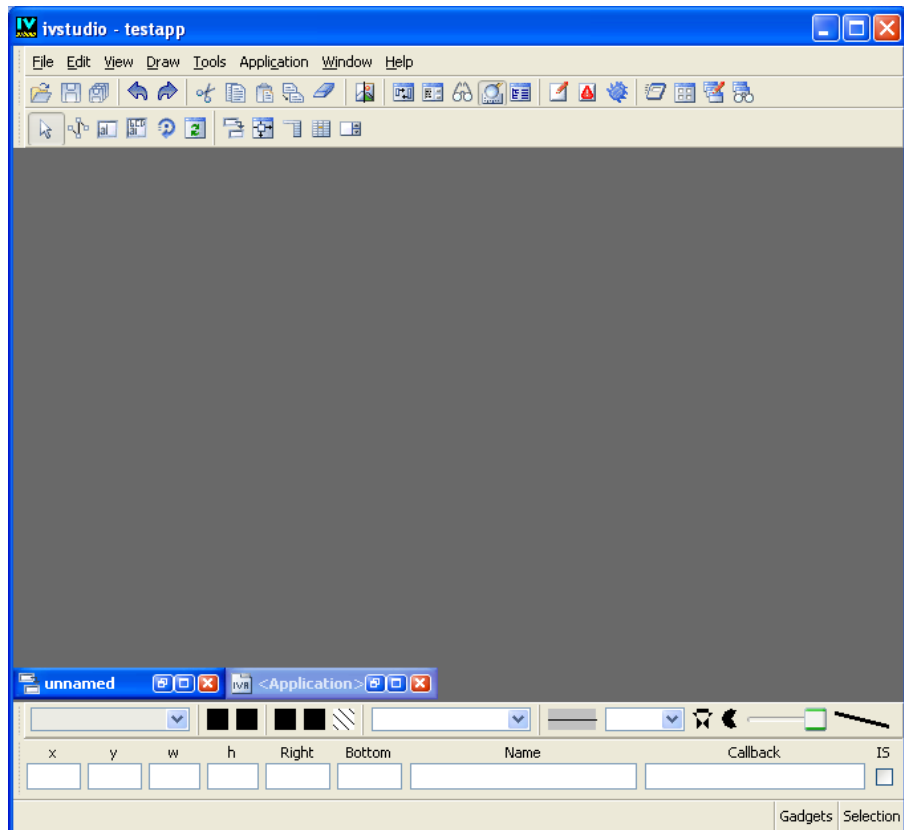


Figure 16.8 Horizontal Toolbar

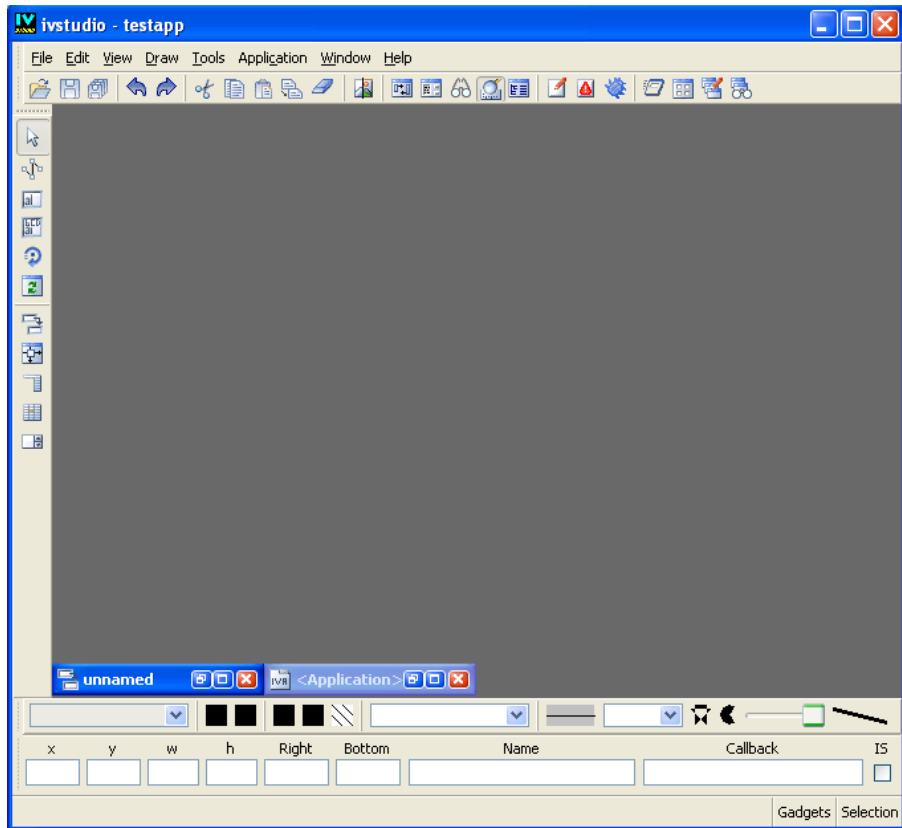


Figure 16.9 Vertical Toolbar

Note: This class manages its own subclass of `IlvDockable` and, therefore, must not be modified.

Customizing Docking Bars

The `IlvAbstractBarPane` class has virtual member functions that you can redefine to meet your specific needs:

- ◆ `IlvAbstractBarPane::orientationChanged`—Is called each time the orientation of the toolbar encapsulated by the pane changes.
- ◆ `IlvAbstractBarPane::geometryChanged`—Is called each time the geometry of the toolbar encapsulated by the pane changes. See *Notifying the Bar About Geometry Changes* on page 299.

The following example shows a subclass of the `IlvAbstractBarPane` class that changes the orientation of the labels according to the bar orientation:

```
class MyMainMenuBarPane
: public IlvAbstractBarPane
{
public:
    MyMainMenuBarPane(const char* name, IlvAbstractBar* bar)
        : IlvAbstractBarPane(name, bar) {}
    virtual void setContainer(IlvPanedContainer* container)
    {
        IlvAbstractBarPane::setContainer(container);
        if (container)
            checkLabelOrientation();
    }
    virtual void orientationChanged()
    {
        checkLabelOrientation();
        IlvAbstractBarPane::orientationChanged();
    }
    void checkLabelOrientation()
    {
        IlvDockable* dockable = IlvDockable::GetDockable(this);
        getBar()->setLabelOrientation(dockable && dockable->isDocked()
                                     ? getBar()->getOrientation()
                                     : IlvHorizontal,
                                     IlFalse,
                                     IlFalse);
    }
};
```

The `checkLabelOrientation` member function is called each time the bar orientation changes. It sets the orientation of the bar labels to the bar orientation if the pane is docked, or to `IlvHorizontal` if the bar is undocked.

Building a Standard Application With Docking Panes

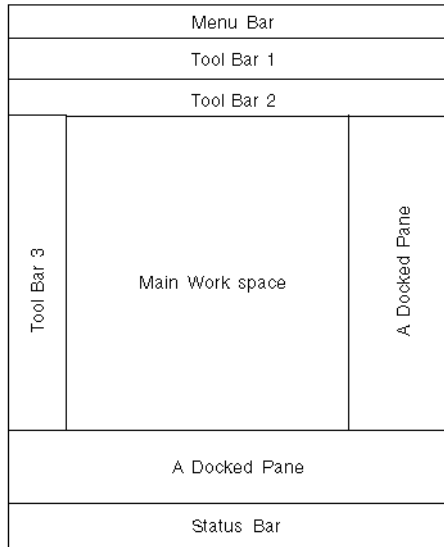
GUI applications with docking panes all have more or less the same look. IBM® ILOG® Views Gadgets provides a class that lets you build standard GUI applications with docking panes very easily.

This section covers the following topics:

- ◆ *Defining a Standard Layout*
- ◆ *Using the `IlvDockableMainWindow` Class*

Defining a Standard Layout

As a general rule, standard GUI applications have the following layout:



You can see from the illustration that a standard layout is composed of a central area, called the main workspace, which is surrounded by several panes on the left and right sides and also at the top and bottom.

Here is an example of a typical GUI application with docking panes:

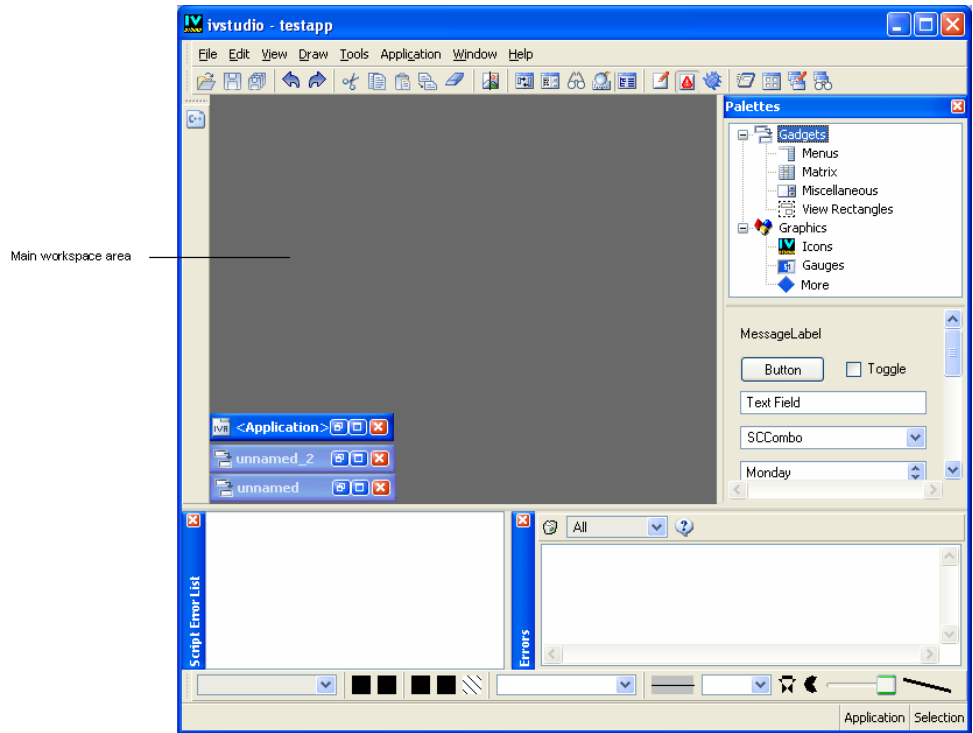
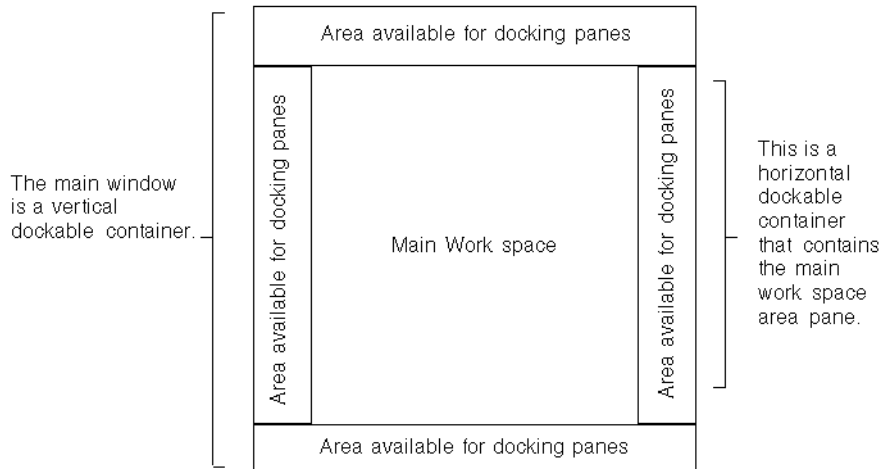


Figure 16.10 Typical GUI Application with Docking Panes

Using the docking pane functionality, you can build a standard GUI application that has the following pane structure:



With this layout, it is possible to add panes anywhere around the main workspace area, as shown on Figure 16.10 on page 344.

Using the `IlvDockableMainWindow` Class

The `IlvDockableMainWindow` class implements the layout described in *Defining a Standard Layout* on page 342. Using this class, you can specify where a pane should be added relative to a specific pane in a very easy way and without knowing exactly how panes are organized. Adding a new pane with the member function

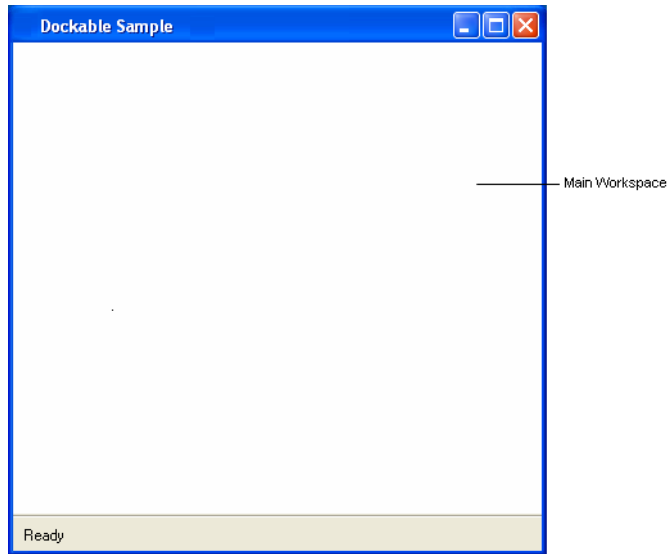
`IlvDockableMainWindow::addRelativeDockingPane` is as simple as using the following sentence to specify where it should go: “I want to put my menu bar on top of the main workspace area.”

Building the whole application interface becomes very simple since what you have to provide is the names of the panes instead of their indexes.

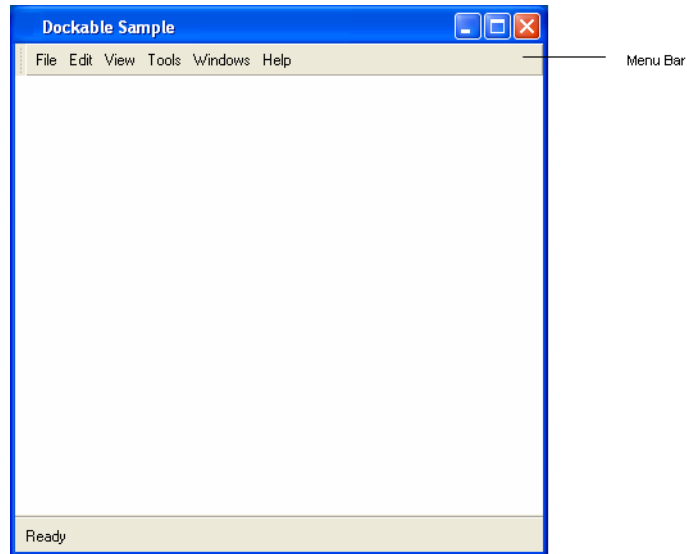
Below is an example of what you can obtain using the member function

`IlvDockableMainWindow::addRelativeDockingPane`.

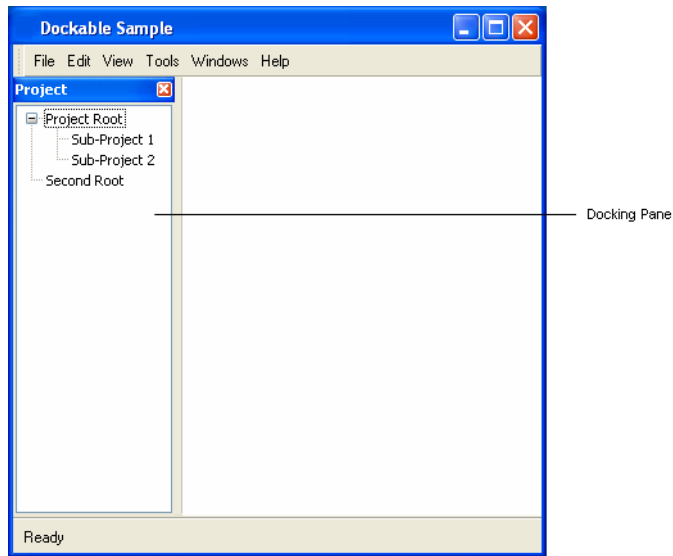
Creating an instance of the `IlvDockableMainWindow` produces the following pane layout:



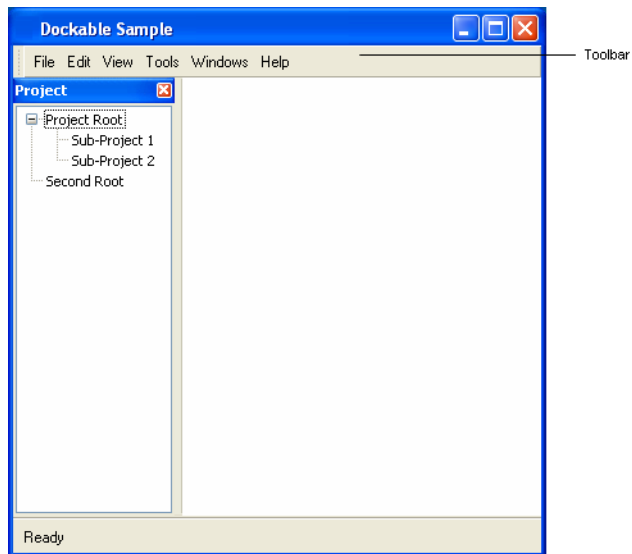
Then, a menu bar is added at the top of the main workspace area, as shown below:



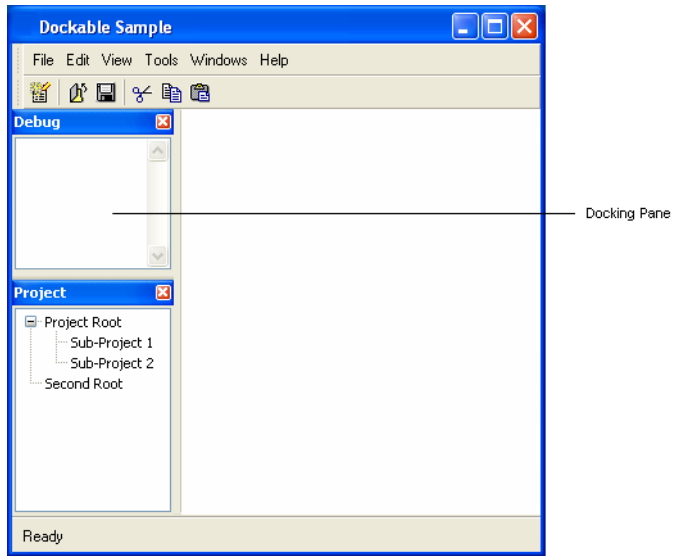
Then a docking pane is added to the left of the main workspace area, as shown below:



Then a toolbar is added underneath the menu bar, as shown below:



Finally a second docking pane is added above the first docking pane, as shown below:



View Frames

The IBM® ILOG® Views Gadgets library supports view frames.

This chapter explains what view frames are and how to use them in your graphical applications. It covers the following topics:

- ◆ *Introducing View Frames*
- ◆ *Creating a Desktop with View Frames*
- ◆ *Managing View Frames*
- ◆ *Minimizing, Maximizing, and Restoring View Frames*
- ◆ *Closing View Frames*
- ◆ *Changing the Current View Frame*

Introducing View Frames

A view frame is a special container with a title bar that encapsulates a client view. Its title bar is composed of an icon, a label, and several buttons. A view frame is displayed inside a parent view, called a desktop view.

View frames are instances of the class `IlvViewFrame`. A desktop view is always linked to an instance of the class `IlvDesktopManager`, which manages all the frames inside the desktop view.

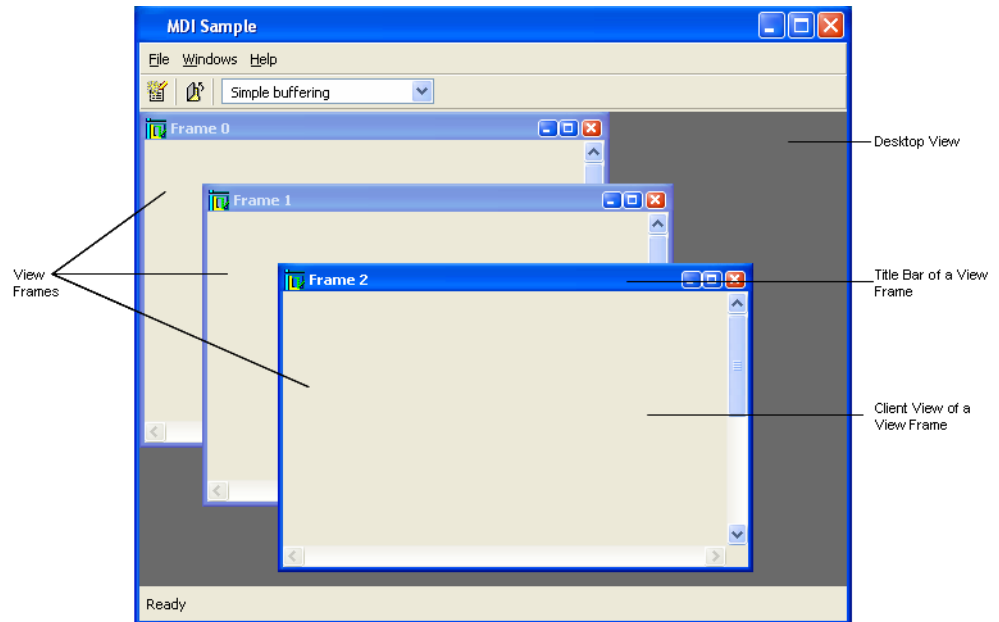


Figure 17.1 Application Composed of Frames

Creating a Desktop with View Frames

This section explains how to create a desktop containing view frames. It covers the following sections:

- ◆ *Creating a Desktop*
- ◆ *Creating View Frames*

Creating a Desktop

The first step consists of creating the desktop view and the desktop manager:

```
IlvView* desktopView = new IlvView(...);  
IlvDesktopManager* desktop = new IlvDesktopManager(desktopView);
```

Note: The desktop view can be an instance of any subclass of `IlvView`.

You can retrieve the `IlvDesktopManager` instance that is linked to a view using the static method `IlvDesktopManager::Get`:

```
IlvDesktopManager* desktop = IlvDesktopManager::Get(view);
```

To know the desktop view associated with a desktop manager, use the method

```
IlvDesktopManager::getView:
```

```
IlvView* view = desktop->getView();
```

Note that when the desktop view is deleted, the desktop manager is notified, but is not deleted.

Creating View Frames

Once the desktop manager is created, you can build the view frames as child windows of the desktop view:

```
IlvViewFrame* vframe = new IlvViewFrame(desktopView,
                                         "Frame 0",
                                         IlvRect(0, 0, 100, 100));
```

The new view frame is automatically managed by the instance of `IlvDesktopManager` that is linked to its parent view (that is, the desktop view). Note that if no desktop manager has been attached to the parent view of a view frame, a default desktop manager is created using the parent view of the view frame as desktop view. This default desktop manager is internally managed, so you will not have to delete it.

To know the desktop manager of a view frame, use:

```
IlvDesktopManager* desktop = vframe->getDesktopManager();
```

You can also retrieve the list of frames managed by a desktop manager using:

```
IlvUInt count;
IlvViewFrame* const* frames = desktop->getFrames(count);
```

Managing View Frames

This section covers the following topics:

- ◆ *Creating a Client View*
- ◆ *Changing the Title Bar*

◆ Changing the View Frame Menu

Creating a Client View

When created, a view frame has no client view. To add a client view to a view frame, you must create a child window inside it:

```
IlvGadgetContainer* clientView =  
    new IlvGadgetContainer(vframe, IlvRect(0, 0, 200, 200));
```

The view frame is resized to fit the client view geometry.

Note: The client view can be any instance of any subclass of `IlvView`.

To know the client view associated with a view frame, use:

```
IlvView* clientView = vframe->getClient();
```

Note: A view frame should only handle one client view.

Changing the Title Bar

The title bar consists of an icon, a title, and three buttons.



Figure 17.2 The Title Bar of a View Frame

To change the icon of the title bar, use the method `IlvViewFrame::setIcon`:

```
IlvBitmap* bitmap = ...  
vframe->setIcon(bitmap);
```

To change its title, use the method `IlvViewFrame::setTitle`:

```
vframe->setTitle("Frame Title");
```

The three buttons to the right of the title bar are used to switch to one of the three states that a frame can have. These are detailed below.

Changing the View Frame Menu

Each view frame has a pop-up menu that is displayed when you click the icon located on the left end of the title bar.

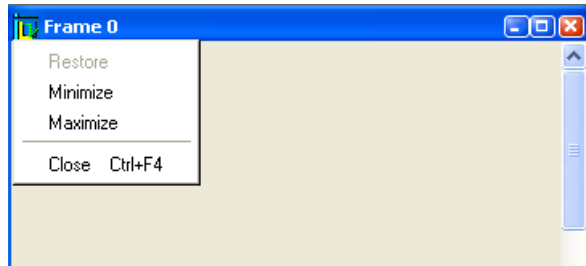


Figure 17.3 *The Pop-up Menu of a View Frame*

By default, the pop-up menu of a view frame contains the following choices: Restore, Minimize, Maximize, and Close. You can however add new items to it.

To access this menu, use:

```
IlvPopupMenu* popup = vframe->getMenu();
```

Minimizing, Maximizing, and Restoring View Frames

A view frame can be in one of the following states: Normal, Minimized, Maximized.

To retrieve the state of a frame, use the method `IlvViewFrame::getCurrentState`. The possible returned values are: `NormalState`, `MinimizedState`, and `MaximizedState`.

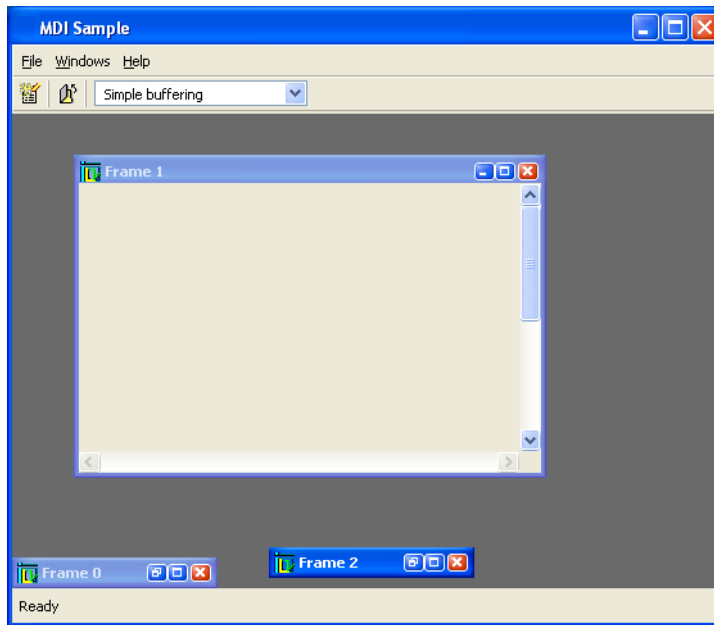


Figure 17.4 Normal and Minimized View Frames

Normal View Frames

By default, a view frame is displayed with its normal size. To restore a frame to this state after it has been maximized or minimized, use the `IlvViewFrame::restoreFrame` method:

```
vframe->restoreFrame();
```

This method does nothing if the frame is already in the normal state.

You can also revert a view frame to its initial state by clicking the Restore button in its title bar.

Minimized View Frames

When a view frame is minimized, only its title bar is visible, and its position is managed by the desktop manager. To minimize a frame, use the `IlvViewFrame::minimizeFrame` method:

```
vframe->minimizeFrame();
```

A list of minimized view frames is managed by the desktop manager, and can be accessed using the `IlvDesktopManager::getMinimizedFrames` method.

You can also minimize a view frame by clicking the Minimize button in its title bar.

Maximized View Frames

When a view frame is maximized, its client view occupies all the desktop view.

To maximize a view frame, use the method `IlvViewFrame::maximizeFrame`:

```
vframe->maximizeFrame();
```

You can also maximize a view frame by clicking the Maximize button in its title bar.

When a view frame is maximized, its title bar and hence the buttons it contains are no longer visible. In this case, however, the desktop manager can display these buttons in another place.

The following lines tell the desktop manager to display the buttons of the title bar in container when a frame is maximized.

```
IlvContainer* container = ...
desktop->makeMaximizedStateButtons(container->getHolder());
```

The following lines tell the desktop manager to display the buttons of the title bar in toolbar:

```
IlvToolBar* toolbar = ...
desktop->makeMaximizedStateButtons(toolbar);
```

Closing View Frames

When you try to close a view frame (using the Close button for example), the `IlvViewFrame::closeFrame` method is called. By default, this method invokes the destroy callbacks set for the view frame, which means that if you want to control how a view frame is destroyed, you have to set a destroy callback.

For example:

```
vframe->setDestroyCallback(DestroyFrame);
```

with the following callback:

```
static void DestroyFrame(IlvView* view, IlvAny)
{
    IlvIQuestionDialog dlg(view->getDisplay(), "Are you sure ?");
    dlg.moveToMouse();
    if (dlg.get())
        delete view;
}
```

displays a dialog box asking confirmation to the user before deleting the frame.

Note: *If no destroy callback has been set, attempting to close the view frame has no effect.*

Changing the Current View Frame

The current view frame of a desktop manager is the view frame that has the keyboard focus. You can change the current view frame by clicking another view frame, which will become the new current view frame.

You can also change the current view frame by coding:

```
desktop->setCurrentFrame(vframe);
```

When the current view frame changes, the virtual method

`IlvDesktopManager::frameSelectionChanged` is called. You can override this method in your own subclass of `IlvDesktopManager` to execute a specific action when the current view frame changes.

Customizing the Look and Feel

This chapter introduces the classes used by the look-and-feel mechanism. It covers the following topics:

- ◆ *Understanding the Architecture*
- ◆ *Making a User-Defined Component Look-and-Feel Dependant*
- ◆ *Changing the Look and Feel of an Existing Component*
- ◆ *Creating a New Look-and-Feel Handler*

Understanding the Architecture

The purpose of this section is to explain how gadgets can adapt themselves to their look and feel. You can find information on the following topics:

- ◆ *IlvLookAndFeelHandler*
- ◆ *IlvObjectLFHandler*
- ◆ *Class Diagram*

IlvLookAndFeelHandler

The `IlvLookAndFeelHandler` class is the base class for all the look-and-feel handlers. It acts as a collection of object look-and-feel handlers and gathers properties common to a specific look. Each component that needs to be look-and-feel dependant must have an access to an instance of the `IlvLookAndFeelHandler` class. During the drawing process, the component will use this handler to draw itself. Similarly, when the component receives an event, it will use this handler to handle the event the way it is defined by the handler.

Note: *Look-and-feel handlers are shared objects. You should not create them using the standard operator `new`, and you should not delete them.*

Getting a Pointer to an IlvLookAndFeelHandler Object

There are three ways for a gadget to get a pointer to an `IlvLookAndFeelHandler` subclass instance:

- ◆ Object level

The method `IlvGraphic::getLookAndFeelHandler()` is used to query an object about its look-and-feel handler. The default implementation is to use the look-and-feel handler defined by the object holder.

- ◆ Holder level

The method `IlvGraphicHolder::getLookAndFeelHandler()` is used to query a holder about its look-and-feel handler. The default implementation is to use the look-and-feel handler defined by the holder display instance.

- ◆ Display level

The method `IlvDisplay::getLookAndFeelHandler()` is used to query a display instance about its look-and-feel handler. The default value is defined by the platform on which the application has been built. See the section *Using the Default Look and Feel* on page 218 for details.

IlvObjectLFHandler

Once a gadget has retrieved its look-and-feel handler, it must ask its specific object look-and-feel handler. This object look-and-feel handler is implemented by means of the `IlvObjectLFHandler` class. Each component that needs to be look-and-feel dependant must create a subclass of the `IlvObjectLFHandler` class.

Getting a Pointer to an IlvObjectLFHandler Object

The `IlvLookAndFeelHandler` class handles a hash table of `IlvObjectLFHandler`. Each instance of the `IlvObjectLFHandler` class can be retrieved using its class information.

For example, the following code retrieves the object look-and-feel handler of the `IlvButton` class:

```
IlvLookAndFeelHandler lfh = display->getLookAndFeelHandler();
IlvButtonLookAndFeelHandler* buttonLF = (IlvButtonLookAndFeelHandler*)
    lfh->getObjectLookAndFeelHandler(IlvButton::ClassInfo());
```

Note: *The value returned by `getObjectLookAndFeelHandler` is cast into an `IlvButtonLookAndFeelHandler` pointer, which is the base class for button object look-and-feel handlers.*

After retrieving a pointer to its specific object look-and-feel handler, the button can draw itself using the following code:

```
void
IlvButton::draw(IlvPort* dst,
               const IlvTransformer* t,
               const IlvRegion* clip) const
{
    IlvButtonLookAndFeelHandler* lfhandler = (IlvButtonLookAndFeelHandler*)
        getObjectLookAndFeelHandler(IlvButton::ClassInfo());
    lfhandler->draw(this, dst, t, clip);
}
```

Class Diagram

The following diagram shows the relations between the three actors of the look-and-feel process: the objects, the look-and-feel handler, and the object look-and-feel handlers.

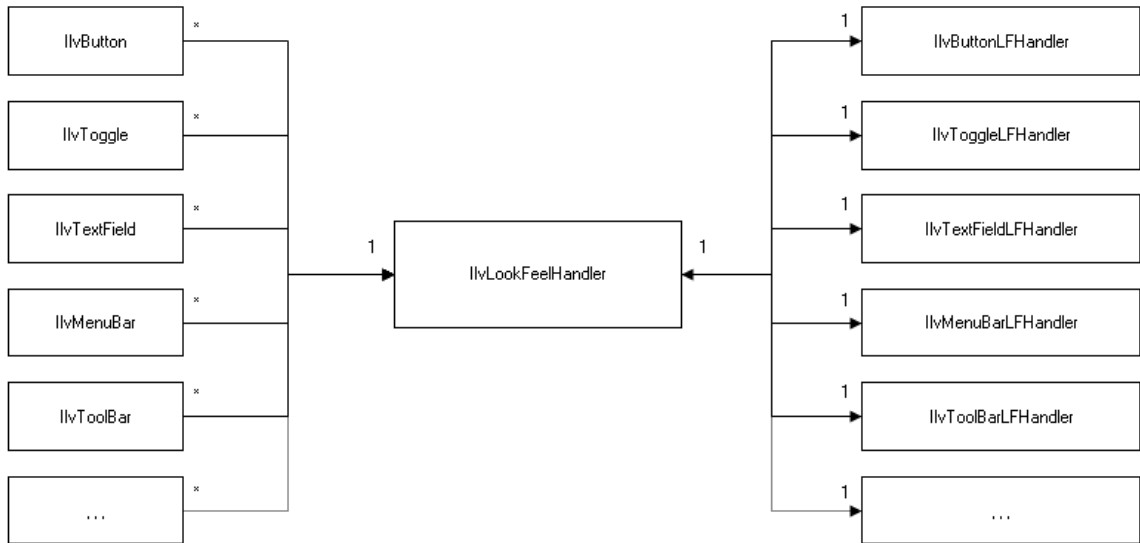


Figure 18.1 Relations between some classes involved in the look-and-feel process

The following diagram is a trace of events during the drawing of an `IlvButton`, in Motif look:

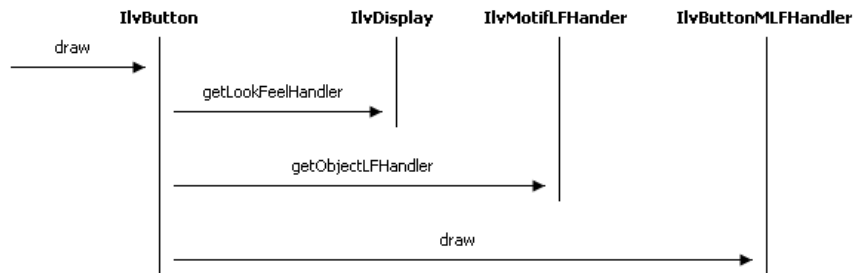


Figure 18.2 Event Trace: Drawing a button

Making a User-Defined Component Look-and-Feel Dependant

This section describes how to create a new component that will be look-and-feel dependant. You will find information on the following topics:

- ◆ *Creating a New Component*
- ◆ *Defining the Object Look-and-Feel Handler API*
- ◆ *Subclassing the Object Look-and-Feel Handler*
- ◆ *Installing the Object Look-and-Feel Handlers*

Creating a New Component

You can find detailed information on how to create properly a new component in the Foundation User's Manual, *IlvGraphic: The graphic object class, Creating a New Graphic Object Class*.

The key is to register properly the class information, which is mandatory to make the component look-and-feel dependant.

Let's assume that the new created component is `MyComponent`, a subclass of `IlvGadget`.

Defining the Object Look-and-Feel Handler API

The object look-and-feel handler API depends on the component you are designing. As a general rule, you should provide a way to customize its look and its behavior. This can be done by adding the following methods to your object class:

```
class MyComponent : public IlvGadget
{
    ...
    virtual void draw(IlvPort* dst,
                    const IlvTransformer* t,
                    const IlvRegion* clip) const;
    virtual IlBoolean handleEvent(IlvEvent& event) const;
    ...
};
```

You must also add these methods to the object look-and-feel handler class:

```
class MyComponentLFHandler : public IlvObjectLFHandler
{
    MyComponentLFHandler(IlvLookFeelHandler* lfh) :
        IlvObjectLFHandler(MyComponent::ClassInfo(), lfh) {}
    ...
    virtual void draw(const MyComponent* object,
                    IlvPort* dst,
                    const IlvTransformer* t,
                    const IlvRegion* clip) const = 0;
    virtual IlBoolean handleEvent(MyComponent* object,
                                IlvEvent& event) const = 0;
    ...
};
```

Notes:

1. Since object look-and-feel handlers are shared objects, you need to give access to `MyComponent` instance in each method of the object look-and-feel handler class. You can do this by using the first parameter of the methods.
2. The constructor of `MyComponentLFHandler` uses the `MyComponent::ClassInfo` method to link this object handler with `MyComponent` class. Thus, each subclass of `MyComponentLFHandler` will be dedicated to `MyComponent` component.

The implementation of the `MyComponent` methods should be as follow:

```
void
MyComponent::draw(IlvPort* dst,
                 const IlvTransformer* t,
                 const IlvRegion* clip) const
{
    MyComponent* lfhandler = (MyComponentLFHandler*)
        getObjectLFHandler(MyComponent::ClassInfo());
    lfhandler->draw(this, dst, t, clip);
}
```

and for the `handleEvent` method:

```
IlvBoolean
MyComponent::handleEvent(IlvEvent& event)
{
    MyComponentLFHandler* lfhandler = (MyComponentLFHandler*)
        getObjectLFHandler(ClassInfo());
    return lfhandler->handleEvent(this, event);
}
```

Of course, you can add other functionalities to your component, and make them look-and-feel dependant using the same scheme.

Subclassing the Object Look-and-Feel Handler

Once the API of the object look-and-feel handler has been defined, you can implement various subclasses corresponding to different look-and-feel styles. For example, here we

create the subclass of `MyComponentLFHandler` dedicated to the Motif look, the `MyComponentMLFHandler` class:

```
class MyComponentMLFHandler : public MyComponentLFHandler
{
    MyComponentMLFHandler(IlvLookAndFeel* lfh) :
        MyComponentLFHandler(lfh) {}
    ...
    virtual void draw(const MyComponent* object,
                     IlvPort* dst,
                     const IlvTransformer* t,
                     const IlvRegion* clip) const;
    virtual IlvBoolean handleEvent(MyComponent* object,
                                   IlvEvent& event) const;
    ...
};
```

You need now to install your object look-and-feel handlers so that they will be used when the corresponding look and feel is set.

Installing the Object Look-and-Feel Handlers

To install your object look-and-feel handlers on their corresponding look-and-feel handler, use the macro `IlvRegisterObjectLFHandler`:

```
IlvRegisterObjectLFHandler(IlvMotifLFHandler,
                           MyComponent,
                           MyComponentMLFHandler);
```

The previous code registers the `MyComponentMLFHandler` class as the object look-and-feel handler for the `MyComponent` class displayed using the Motif look.

You do not have to create or delete instances of object look-and-feel handlers, it will be done automatically.

Changing the Look and Feel of an Existing Component

This section describes how to modify the look and feel of a specific component. You can find information on the following topics:

- ◆ *Subclassing the Component Object Look-and-Feel Handler*
- ◆ *Replacing an Object Look-and-Feel Handler*

Subclassing the Component Object Look-and-Feel Handler

To change the look and feel of a component, you must first identify its object look-and-feel handler base class. Usually, the component class and its object look-and-feel handler are declared in the same header file, and the name of the object look-and-feel handler class is the

concatenation of the component class name with the string “LFHandler”. For example, both the `IlvButton` and `IlvButtonLFHandler` classes are located in the `<ilviews/gadgets/button.h>` header file.

Once you have found the object look-and-feel handler base class, you must look closely at its API to find which of the virtual member functions need to be overridden.

The following example is a subclass of the `IlvButtonLFHandler` where the `drawBackground` member function has been redefined:

```
class MyButtonLFHandler : public IlvButtonLFHandler
{
    ...
    virtual void drawBackground(const IlvButton* button,
                               IlvPort* dst,
                               const IlvTransformer* t,
                               const IlvRegion* clip) const;
    ...
};
```

Replacing an Object Look-and-Feel Handler

Once you have defined the new object look-and-feel handler, you need to install it on an `IlvLookFeelHandler` instance.

The simplest way to install an object look-and-feel handler is to call the `IlvLookFeelHandler::addObjectLFHandler` method on the look-and-feel handler of the component:

```
IlvButton* button = ...
IlvLookFeelHandler* lfh = button->getLookFeelHandler();
MyButtonLFHandler* mylfh = new MyButtonLFHandler(lfh);
lfh->addObjectLFHandler(mylfh);
```

By modifying the look-and-feel handler this way will affect other buttons referencing the same look-and-feel handler. Indeed, by default, there is only one look-and-feel handler, owned by the `IlvDisplay` class. If you do not want to modify the default look-and-feel handler because you want to modify only the look and feel of specific components, you must do the following:

- ◆ Create a new look-and-feel handler instance using the `IlvLookFeelHandler::Create` method.
- ◆ Install your object look-and-feel handler using the `IlvLookFeelHandler::addObjectLFHandler` method.
- ◆ Install the new look-and-feel handler instance on your component using the `IlvGadget::setLookFeelHandler` method.

The following code creates a new look-and-feel handler for the Motif look, and installs on it the object look-and-feel handler. Finally, the new look-and-feel handler is installed on the component:

```
IlvLookAndFeel* lfh = IlvLookAndFeel::Create("motif");
MyButtonLFHandler* mylfh = new MyButtonLFHandler(lfh);
lfh->addObjectLFHandler(mylfh);
IlvButton* button = ...
button->setLookAndFeel(lfh);
```

Note: The first two steps can be executed through a single action by creating a subclass of the `IlvLookAndFeel`. You can find more information in the section *Creating a New Look-and-Feel Handler on page 366*.

Creating a New Look-and-Feel Handler

To create a new look-and-feel handler, you can either:

- ◆ subclass directly the `IlvLookAndFeel` class or
- ◆ subclass one of the existing predefined look-and-feel handler classes (`IlvMotifLFHandler`, `IlvWindowsLFHandler`, `IlvWindows95LFHandler`, `IlvWindowsXPLFHandler`).

The second option is easier, since you do not have to provide an object look-and-feel handler for all the registered gadgets. You just have to provide the object look-and-feel handlers for the objects you want the look and feel to be changed.

The sample `lookfeel` located in the directory `ILVHOME/samples/gadgets/lookfeel` shows how to create a new look-and-feel handler.

Registering a New Look-and-Feel Handler

To be able to dynamically create a look-and-feel handler, you need to properly register it. To do this, you need to add the following macro inside the class declaration:

```
DeclareLookAndFeelTypeInfo();
```

Then, in the definition file, use the following macros:

```
IlvPredefinedLookAndFeelMembers(MyLookAndFeel, "MyLook");
IlvRegisterLookAndFeelClass(MyLookAndFeel, BaseClass);
```

where `BaseClass` is the base class of the new look-and-feel handler.

The class is now properly registered, that is, you can create for example an instance of it using the following code:

```
IlvLookAndFeelHandler* lfh = IlvLookAndFeelHandler::Create(IlGetSymbol("MyLookAndFeel"));
```

Registering Object Look-and-Feel Handlers Into a New Look-and-Feel Handler

To register object look-and-feel handlers into a new look-and-feel handler, you can either:

- ◆ override the `IlvLookAndFeelHandler::createObjectLFHandler` method or
- ◆ use the `IlvRegisterObjectLFHandler` macro.

For example, by using the `IlvRegisterObjectLFHandler` macro, you can code in the definition file:

```
IlvRegisterObjectLFHandler(MyLookAndFeelHandler, IlvButton, MyButtonLFHandler);
```

This will register the object look-and-feel handler class `MyButtonLFHandler` into the look-and-feel handler `MyLookAndFeelHandler` for the `IlvButton` class. This means that when an `IlvButton` object that has the look-and-feel `MyLookAndFeelHandler` tries to retrieve its object look-and-feel handler, it will get a pointer on a `MyButtonLFHandler` instance.

Part III

IBM ILOG Views Application Framework

This part describes the Application Framework package of IBM® ILOG® Views Controls. Application Framework is a library designed to simplify the task of developing your graphical user interface (GUI) for applications based on the IBM ILOG Views Component Suite of C++ libraries for graphics creation and control.

This part contains the following chapters:

- ◆ Chapter 19, *Introducing IBM ILOG Views Application Framework* provides an overview of the document/view architecture and other features of the Application Framework package of ILOG Views.
- ◆ Chapter 20, *Using the Application Framework Editor* describes the Application Framework Editor, itself an easy-to-use GUI.
- ◆ Chapter 21, *Implementing an Application* provides the steps and classes necessary to incorporate the document/view architecture of Application Framework.
- ◆ Chapter 22, *Application Framework Interfaces* describes how to incorporate the interface mechanism of Application Framework.

- ◆ Chapter 23, *Actions* describes how to activate and process actions under Application Framework.

Introducing IBM ILOG Views Application Framework

The IBM® ILOG® Views Application Framework provides an easy-to-use graphics user interface (GUI) for defining the user interface for an application. This chapter provides an overview of the IBM ILOG Views Application Framework package. It includes the sections:

- ◆ *What is Application Framework*
- ◆ *The Document/View Architecture*

What is Application Framework

Application Framework is a library that lets you develop complete applications, such as the one shown in Figure 19.1:

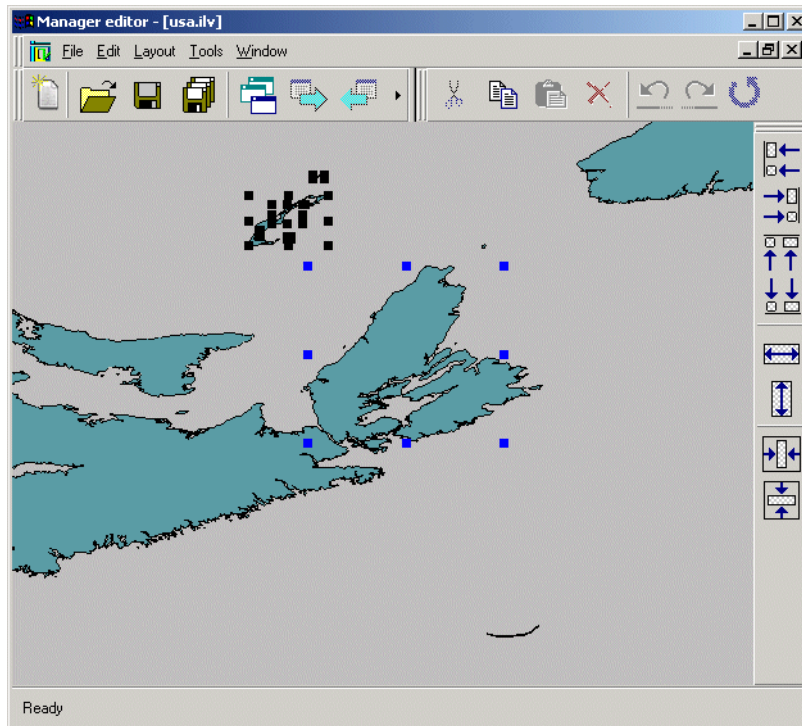


Figure 19.1 Application Developed with Application Framework

It provides a tool called the Application Framework Editor, which allows you to edit the application graphically: all menus, bars, actions, dynamic menus, and so on, are specified using this tool. Chapter 2 describes in detail how to start up and use the Application Framework Editor.

Application Framework also provides a mechanism that allows its objects to track and process GUI events. This mechanism will be looked at in Chapter 22, *Application Framework Interfaces*.

The Document/View Architecture

Application Framework is built on a Document/View architecture, common to most Windows applications. In this type of architecture, the application is a *frame window* holding toolbars and menus, that allows you to edit several documents at the same time. This frame window manipulates *documents* (data that is opened using menu items such as File > Open, File > New, and so on) that the user can edit inside *views*, which are usually created in a frame window.

For example, a document in Microsoft Excel is a table in memory loaded from an `.xls` file, and the views that can display and modify this document are sheets or charts.

Warning: *In Microsoft applications, the term document is used for both the data in memory and the view that lets the user edit the data.*

Chapter 21, *Implementing an Application* describes the document/view architecture in more detail.

Using the Application Framework Editor

The IBM® ILOG® Views Application Framework provides an easy-to-use graphics user interface (GUI) for defining the user interface for an application.

Starting Up the Application Framework Editor

On initial startup, the Application Framework Editor shows this dialog box:

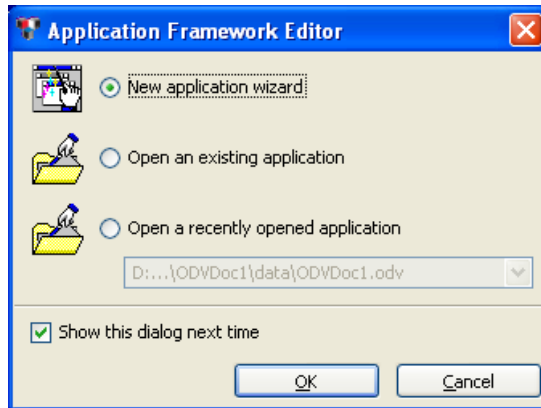


Figure 20.1 The Application Framework Editor Wizard

This dialog box has the following options:

- ◆ New application wizard - to begin creating a new application. See *Creating a New Application*.
- ◆ Open an existing application - to display the browse dialog box for selecting an existing application to open. See *Creating and Configuring an Options File (.odv file)*.
- ◆ Open a recently opened application - to quickly select an application from the drop-down list. See *Application Framework Editor Main Window*.

You can choose to bypass this screen by deselecting the “Show this dialog next time” option. In this case, you are taken directly to the Application Editor main window.

Application Framework Editor Main Window

The Application Framework Editor main window displays a menu bar, action toolbar, status bar, an Application Components palette, and a multidocument workspace. The startup window is shown in Figure 20.2.

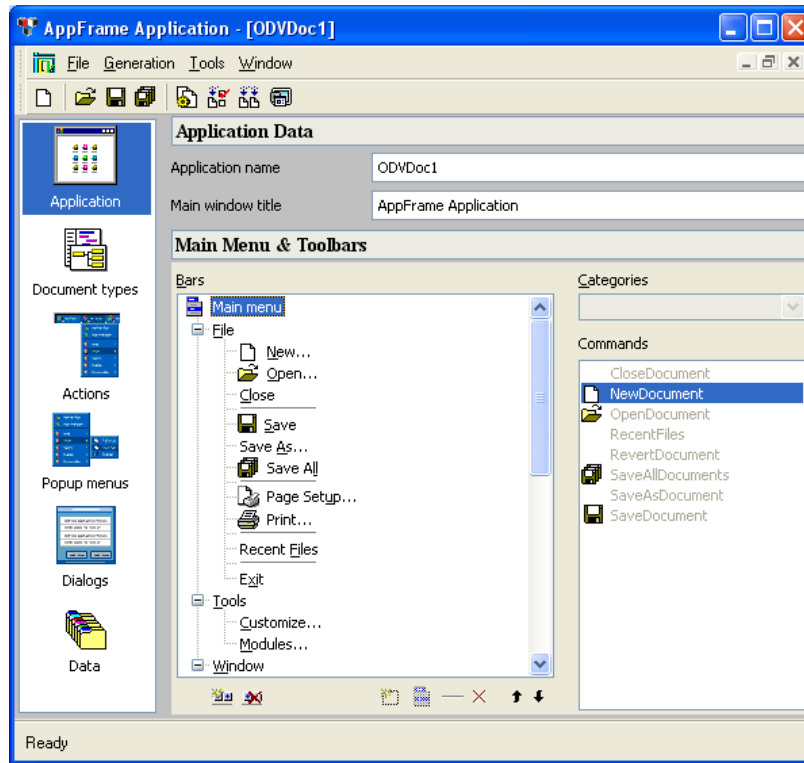


Figure 20.2 Application Editor Main Window

Components Palette

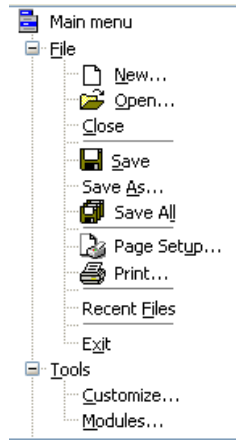
The Application Components palette on the left allows you to select the application entity you are editing:

- ◆ Application
- ◆ Document types
- ◆ Actions
- ◆ Popup menus
- ◆ Dialogs - for dialog boxes and windows
- ◆ Data

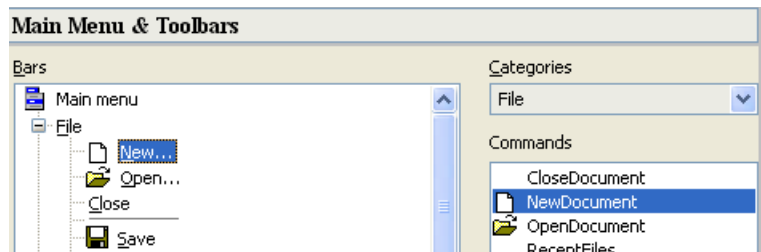
Workspace

The workspace on the right displays and allows you to set the parameters of the selected entry for each of the application entities.

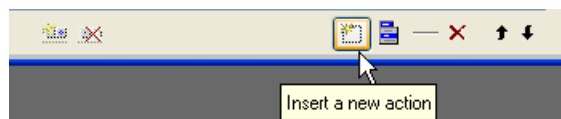
- ◆ A hierarchical tree in the middle area of the workspace allows you to select the item location to be edited or added to. For example:



- ◆ Selecting an item in the tree activates other possible parameter entry fields in the workspace. The fields are tailored to the specific operation. For example:











- ◆ The workspace toolbar at the bottom of the workspace allows you to easily select the operations tailored to the current workspace. For example:



The workspace toolbar changes appropriately, and items are grayed/activated, with the type of application entity. Possible toolbar icons are shown in Table 20.1

Table 20.1 *Workspace Toolbar*

Toolbar Icon	Description
	Insert a new action, popup, or dialog item below the currently selected tree or item; or a new category, accelerator, or other item to a list.
	Insert a new popup menu.
	Insert a new separator
	Remove the currently selected item.
	Move the selected item up in the tree.
	Move the selected item down in the tree.
	Insert a new toolbar.
	Remove the selected toolbar.

Creating a New Application

To develop an application with Application Framework, follow these basic steps:

1. Launch the Application Framework Editor and edit all the application options. Edit the different menus and toolbars that will appear in the application, describe all the document types that the application will be able to open, and so on. These items are saved in an options file that is read by the generated application, when initializing.
2. Generate the application code, using the Application Framework Editor.
3. Complete the generated code:
 - To manage the data (the document):
 - serialize the data,

- add accessors.
- To display and edit the data (the view):
 - initialize the view according to the data (an example of this is filling a tree gadget),
 - manage commands generated by the user events on views.

Note: You can modify the application options described in Step 1 at any time during the development of the application. You are not required to regenerate the code when you modify these options.

The remainder of this chapter describes the general navigation and operational features of the Application Framework Editor. For step-by-step instructions refer to the tutorial sample for Application Framework.

Selecting a Document Type

The first step in creating a new application is to select the document type. When you begin a New application, the Select a document type dialog box appears:

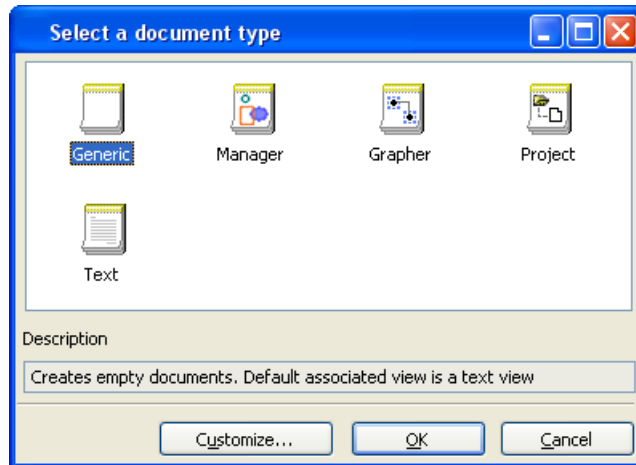







Figure 20.3 Select a Document Type

Several predefined types of documents are made available. Each type of document defines convenient methods for manipulating its data and is preassociated with a specific view.

The document types are described further in Table 20.2.

Table 20.2 Selection of Document Type

Document Type	Use To:	Description
	Create a generic application	The Generic document type does not make any assumptions about the type of document. This is the choice for most applications.
	Create a manager application	The Manager document type deals with <code>IlvGraphic</code> objects inserted in an <code>IlvManager</code> object.
	Create a grapher application	The Grapher document type deals with <code>IlvGraphic</code> objects inserted in an <code>IlvGrapher</code> as nodes or links.
	Create a project application	The Project document type is an organization of files in folders and subfolders.
	Create a text application	The Text document type is any text document.

After you select a document type, the Application Framework Editor main window appears. See *Application Framework Editor Main Window*.

Creating and Configuring an Options File (.odv file)

Application Framework stores all parameters that describe an application in an options file (.odv extension).

The Application Framework Editor opens a new .odv file whenever you create a new application (New from the File menu, toolbar, or initial wizard) and select a document type.

Setting Application Parameters

The Application Framework Editor is used to set your application parameters when *Application* is selected from the Application Framework Editor Palette.

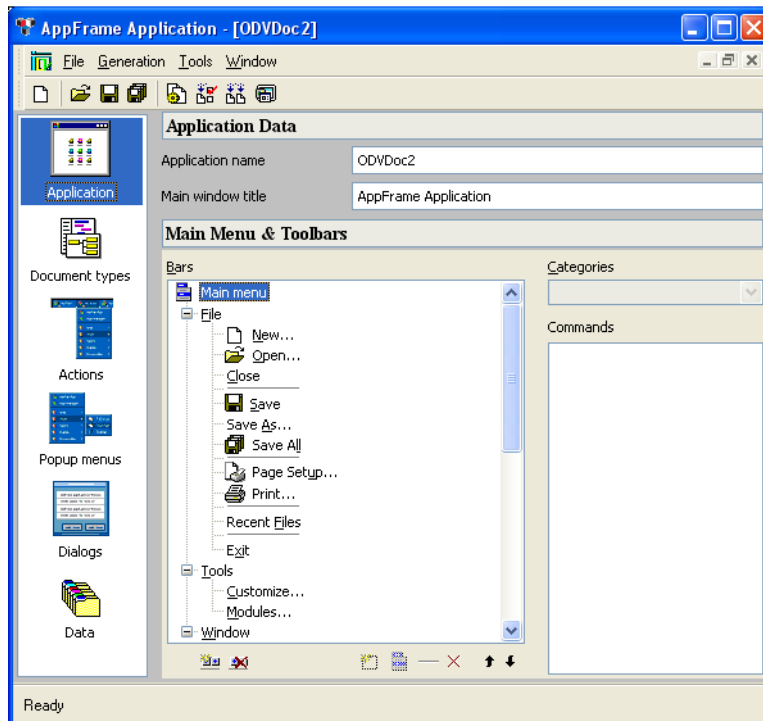



Figure 20.4 Application Selected from Palette

In the Application Data workspace of the Application Framework Editor you set:

- ◆ The application name in the *Application name* field. By default, this name is used to create the directory and the project name. The default name shown in Figure 20.2 is ODVDoc1.
- ◆ The main window title in the *Main window title* field. This is the name that will appear as the title in your application.


Adding Menu Items

You add menu items in the Main Menu & Toolbars section of the workspace when *Application* is selected from the Application Framework Editor Palette.

1. Select an item in the "Main menu" tree where you want to insert a new item. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Categories" combo box and "Commands" list.


If you want to modify a menu item, select the item and modify it by selecting the new action in the "Commands" list.

If you want to insert new commands, refer to *Creating an Action*.

To remove an item in the main menu bar, select the item to remove and click the delete button .


Adding Toolbar Items

You add toolbar items in the Main Menu & Toolbars section of the workspace when *Application* is selected from the Application Framework Editor Palette.

1. Select an item in the "Standard" tree where you want to insert a new button in the toolbar. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Categories" combo box and "Commands" list.

If you want to modify a menu item, select the item and modify it by selecting the new action in the "Commands" list

If you want to insert new commands, refer to *Creating an Action*.

To remove an item from the toolbar, select the item to remove and click the delete button .

Setting Document Parameters

The Application Framework Editor is used to set your document parameters when *Document types* is selected from the Application Framework Editor Palette.

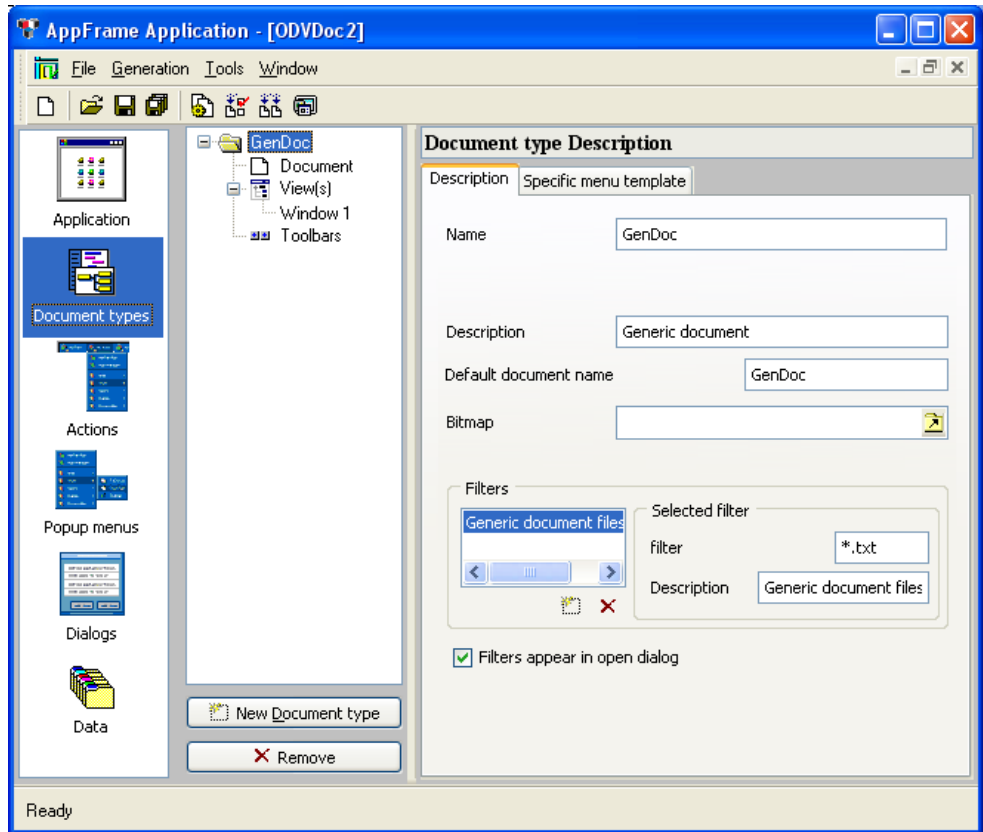



Figure 20.5 'Document Types' Selected from Palette

The middle column shows the document tree, headed by the document type (GenDoc in Figure 20.5; or Grapher, Project, Text, or Manager depending on the chosen type). This column shows the document types that the application can handle. You can add many document types with the "New Document Type" button .

The right column allows you to change parameters of the selected items in the middle columns as described below.

Setting General Document Parameters

When GenDoc is selected in the tree, the workspace has the following tabs:

- ◆ *Description* tab contains basic information about the document type.

The screenshot shows a dialog box titled "Document type Description" with a "Description" tab. The dialog contains the following fields and controls:

- Name:** A text box containing "GenDoc".
- Description:** A text box containing "Generic document".
- Default document name:** A text box containing "GenDoc".
- Bitmap:** An empty text box with a browse button (represented by a folder icon).
- Filters section:**
 - A list box containing "Generic document files".
 - A "Selected filter" field containing "*.*".
 - A "Description" field containing "Generic document files".
 - A checkbox labeled "Filters appear in open dialog" which is checked.

Figure 20.6 'Description' Tab (Document Type Description)

- **Name:** This name is used when you want to retrieve the document template that can create this type of document.
- **Description:** The description of the document type.
- **Default Document Name:** This name is used when a document of this type is created. When created, the document has this name.
- **Bitmap:** This bitmap associated to the document type.
- **Filters section:** This section allows you to define the elements that will appear in the open document dialog box of your application. These elements will be used in the "Files of types" section in the open document dialog box .
- **Filter:** The extension of the document files. The form of this field should be * .xxx where xxx is the extension.
- **Description:** Description of the filter.
- **Filters appears in open dialog:** If the check box is checked, these filters will appear in the open dialog box of your final application.

- ◆ *Specific menu template* tab sets the menu visibility feature for a document.

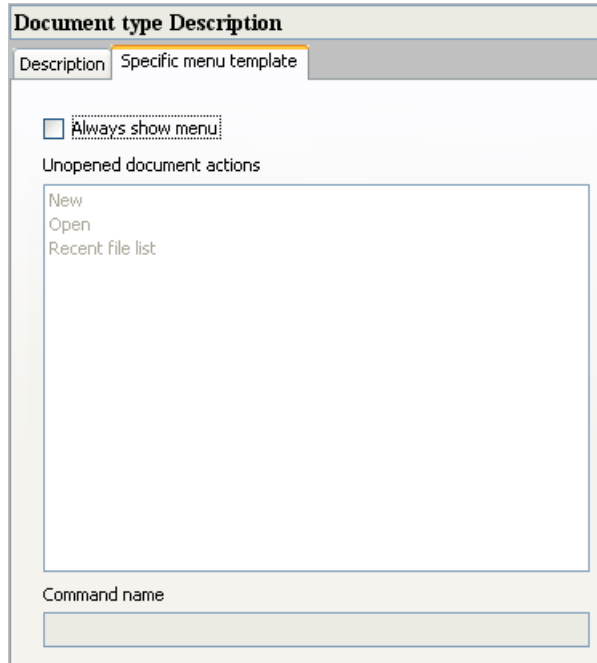


Figure 20.7 'Specific Menu Template' Tab

- ◆ Always show menu: When the toggle is checked, the specific menu and toolbar are added even if a document of this type is not active.

Setting Parameters for a Selected Document

When the `Document` item for a document type is selected in the tree, the workspace has the following fields:

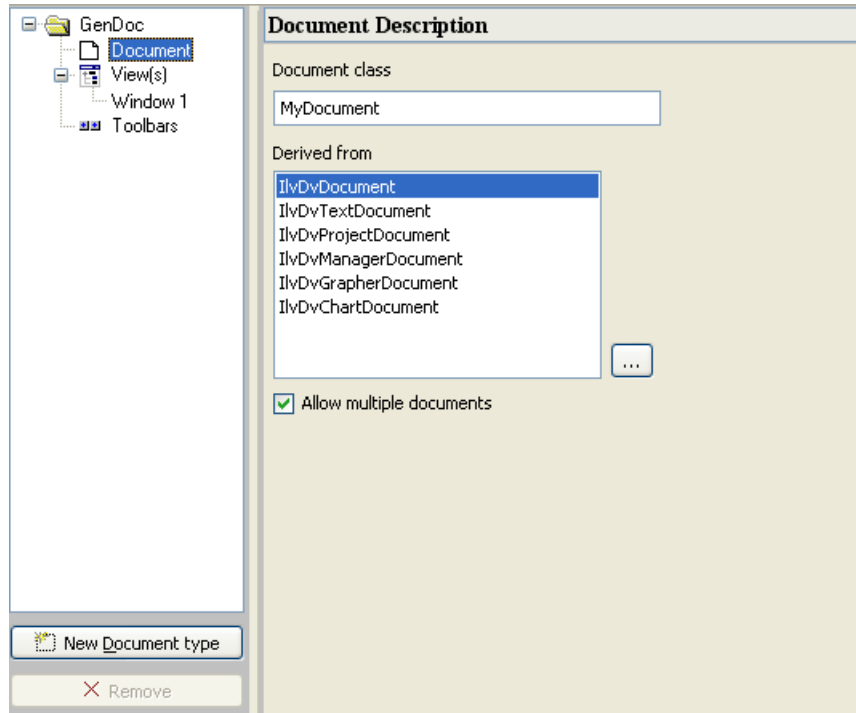


Figure 20.8 'Document Description'

- ◆ The Document class specifies the name of the class used during the code generation.
- ◆ You can specify the parent class by choosing one item in the "Derived from" string list which is filled by a predefined class.
- ◆ The "Allow multiple documents" check box specifies if your application can handle many document of this type or only one.

Setting Window Parameters

When the Window item for a document type is selected in the tree, the workspace has the following tabs:

- ◆ View tab allows you to define the class of the views on the document.

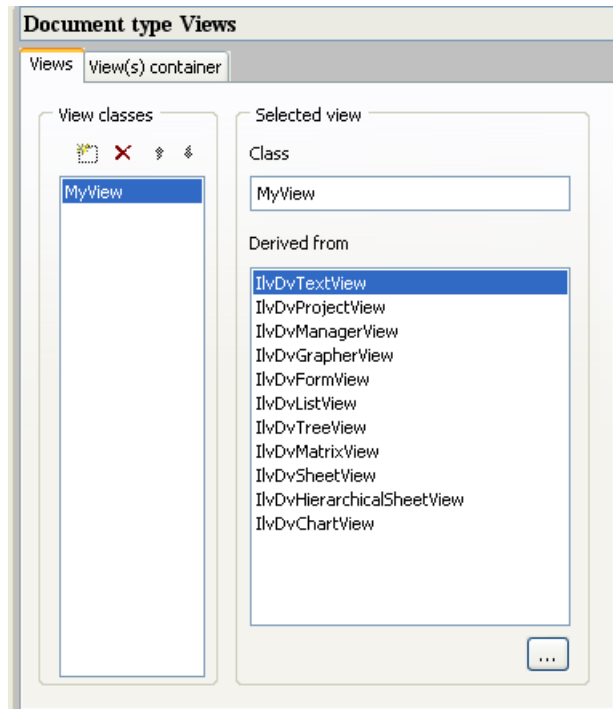


Figure 20.9 'Views' Tab

- In the Class text field, you specify the class name that will be used during the code generation.
- In the "Derived from" list, you select the class from which the selected view class is to be derived.

- ◆ *View(s) container* tab provides additional parameters for the container that will contain the view.

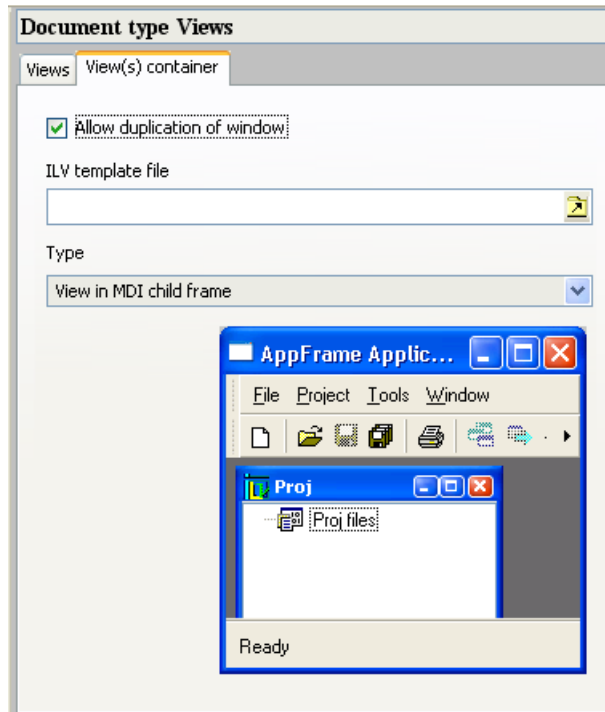


Figure 20.10 'View(s) Container' Tab

- "Allow duplication of window" specifies if your application can handle only one or many views on the same document.
- In the "Type" combo box, you can choose the initialize configuration of your window. You can choose between:
 - View in MDI child frame
 - View in MDI maximized child frame
 - Docked at left
 - Docked at right
 - Docked at top
 - Docked at bottom
 - Docked in a float window

- For all docked configurations, you can specify a method in the "Show/hide action" text field which will be called when this window appears or disappears. This text field is displayed only for docked configurations.

Setting Toolbar Parameters for a Document Type

When the `Toolbars` item is selected in the tree, the workspace allows you to define or change a specific toolbar that is to be displayed only when a document of this type is active. For editing this toolbar see *Adding Toolbar Items*.

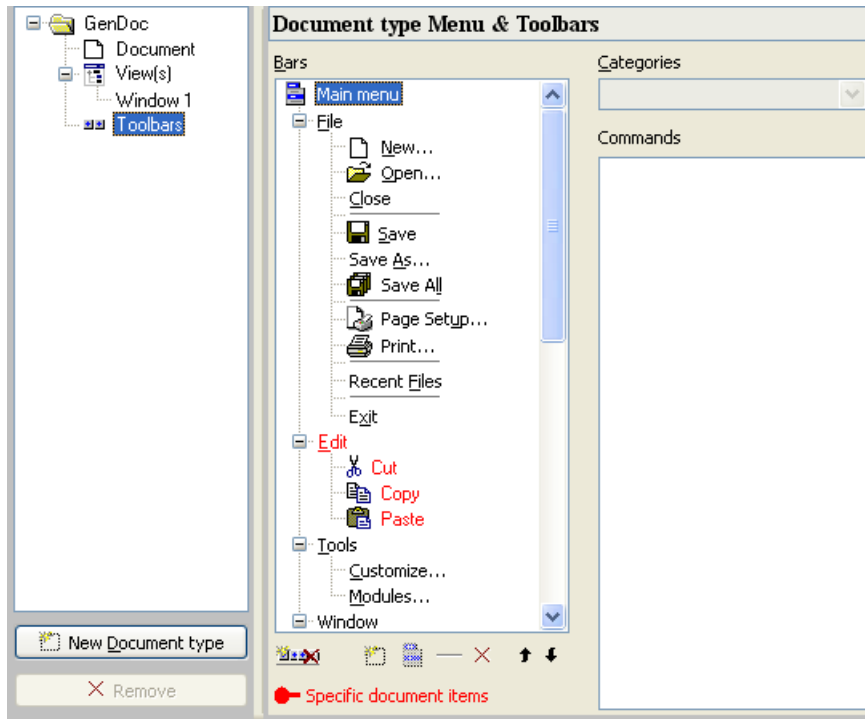


Figure 20.11 'Document Type Menu & Toolbars'

Setting Action Parameters

The Application Framework Editor is used to set your action parameters when *Actions* is selected from the Application Framework Editor Palette.

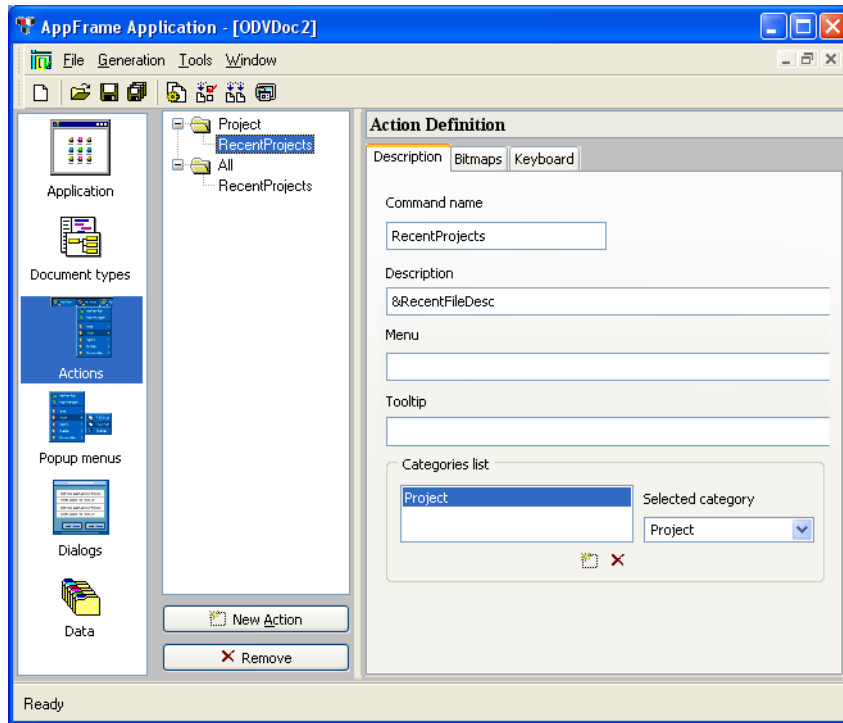


Figure 20.12 'Actions' Selected from Palette

The middle column shows the Actions tree. By clicking either of the `RecentProjects` in the tree, you display the Action Definition.

Action Definition

The Action Definition workspace has the following tabs:

- ◆ *Description* tab contains basic information about the action.

Figure 20.13 ‘Description’ Tab of Action Definition

- **Command Name:** This name is created by the user to identify the action.
- **Description:** The user can enter a description of the action.
- **Menu:** The name of the item appearing in the menu (for example New, Open, Save, and so forth).
- **Tooltip:** The name of the item appearing in the tooltip text (for example “New (Ctrl+N)”).
- **Categories list:** shows the category of the action.
- **Selected Category:** lists the current categories that can be selected (Application, File, Project, and so forth).

The Description, Menu, and Tooltip items can be text, or they can be a message identifier of the form *&identifier* for text contained in a .dbm file. (For information on the .dbm file type see the *IBM ILOG Views Foundation User’s Manual*.)

- ◆ *Bitmaps* tab shows characteristics of the bitmap icon associated with the action. This icon appears in the menu, toolbar, and tree lists with the action name.

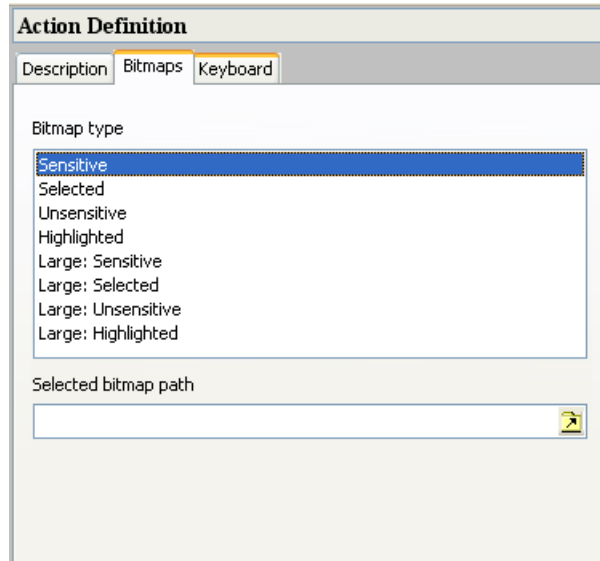


Figure 20.14 'Bitmaps' Tab of Action Definition

- **Bitmap Type:** You can define a set of bitmaps for each different type in the string list. These bitmaps will be used depending on the status of the action.
- **Selected bitmap path:** The path where the icon is found for the selected bitmap type.

- ◆ *Keyboard* tab shows the keyboard shortcut for the action.

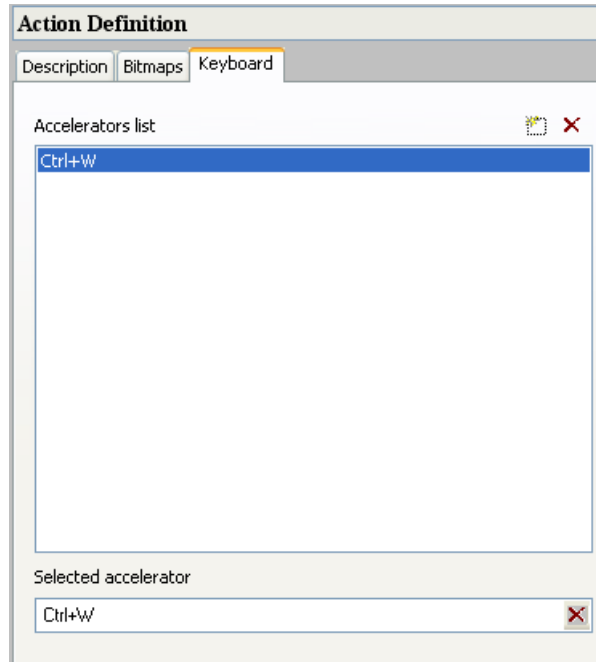



Figure 20.15 'Keyboard' Tab of Action Definition

- Accelerators list: To add an accelerator click the Add button  and use the 'Selected accelerator' field, for example 'Ctrl+W' is added as a default first accelerator.
- Selected Accelerator: The current accelerator or keyboard shortcut. To change the shortcut, click in the field, and then type the sequence of the sortcut on your keyboard.

Creating an Action

For complete details on implementing actions, see Chapter 23, *Actions*.

Setting Popup Menu Parameters

The Application Framework Editor is used to set your popup menu parameters when *Popup menus* is selected from the Application Framework Editor Palette.

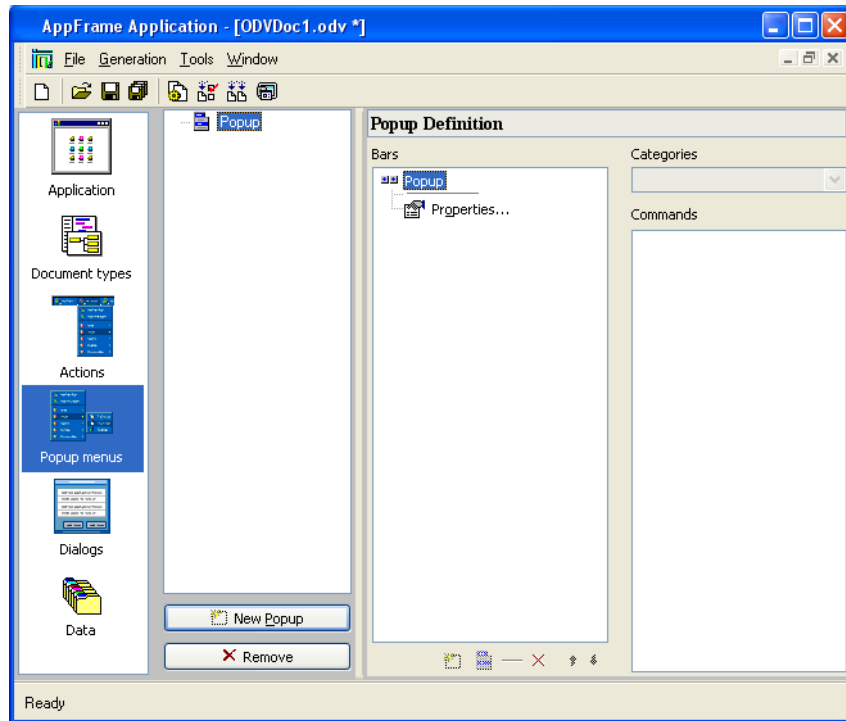


Figure 20.16 'Popup Menus' Selected from Palette

The Popup Definition workspace becomes active when you begin adding a popup menu.

Popup Definition

The Popup Definition workspace allows you to define popups that will be accessible from your application.

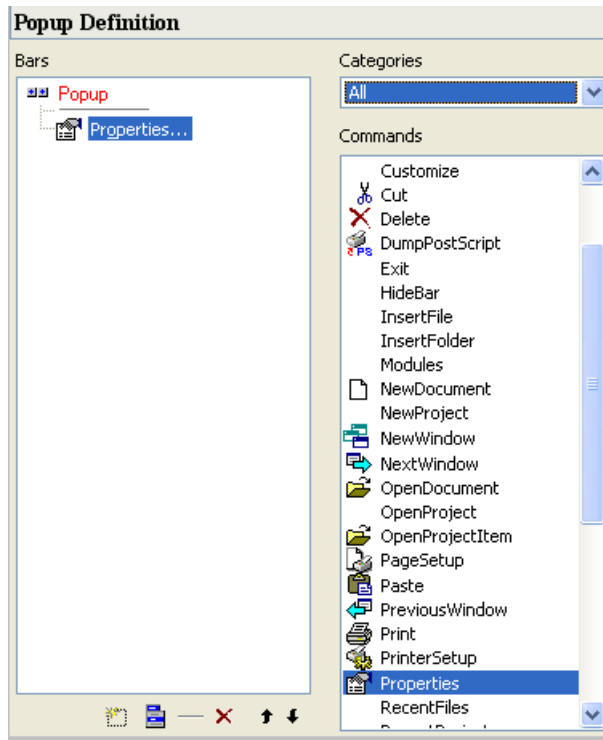



Figure 20.17 Popup Definition

- ◆ The tree shows the layout of the popup menu. When an item of the popup is selected, the remaining fields become activated.
- ◆ Categories: A list showing the possible categories. You can select a category of action to retrieve an action more easily. When the All category is selected (as shown), all commands are displayed alphabetically in the Commands list.
- ◆ Commands: The possible commands in the selected category.

Creating a Popup Menu


To begin defining a new popup menu, click the New Popup button  below the main tree. This column shows a new item in the tree which is the new popup menu created (see Figure 20.16). The created popup has a default layout with two items (separator and Properties items), but you can change this layout.

The default name of the popup menu is `Popupxx` where `xx` is an incremental number when you insert more than one popup menu. You can change the name of this popup by selecting the root item in the Popup Definition window, pressing the F2 key, and then typing the new name.

In your code, you can retrieve this popup by using the `IlvDvApplication::readPopup(const IlvSymbol*)` function.


In the Popup Definition window, you can define the layout of the popup by adding or removing items. For adding items to the popup menu see *Adding a Popup Item*.

Adding a Popup Item

1. Select an item in the popup layout where you want to insert a new item. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Commands" list.


If you want to modify a popup item, select the item and modify it by selecting the new action in the "Commands" list.

If you want to insert new commands, refer to *Creating an Action*.

To remove an item in the popup, select the item to remove and click the delete button .

Adding a New Popup Submenu

The Popup Definition workspace allows you to submenus in the popup.

- ◆ In the Popup Definition tree, select the item where you want to insert the submenu. The submenu will be inserted after this item.
- ◆ Click the "New" popup menu button  in the Popup Definition toolbar.
- ◆ Modify the label of the popup item (use the F2 accelerator) and then enter the new label.
- ◆ Modify the submenu by adding or editing the items of the submenu (see *Adding a Popup Item*).

Setting Dialog Parameters

The Application Framework Editor is used to set your dialog box parameters when *Dialogs* is selected from the Application Framework Editor Palette.

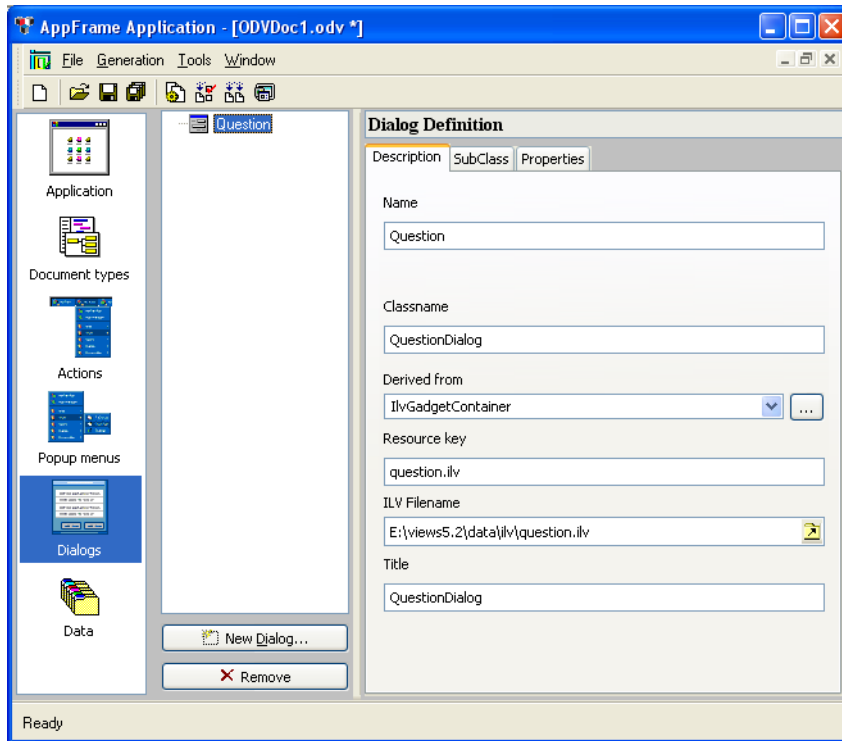


Figure 20.18 'Dialogs' Selected from Palette

The Dialog Definition workspace becomes active when you begin adding a dialog.

Dialog Definition

The Dialog Definition workspace allows you to define dialog box properties. The dialog box must first be defined in IBM ILOG Views Studio.

The Dialog Definition workspace has the following tabs:

- ◆ *Description* tab contains basic information about the dialog box.

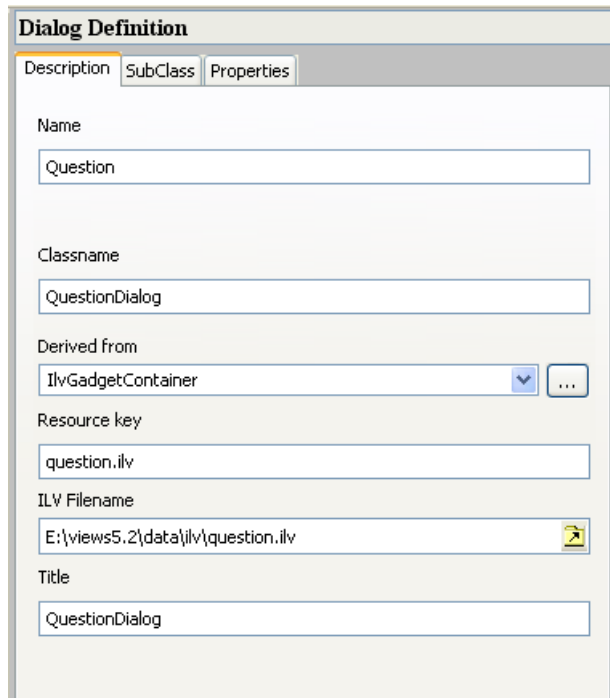


Figure 20.19 'Description' Tab of Dialog Definition

- **Name:** This name is by default the name of the `.ilv` file, for example `Question` when the file loaded is `question.ilv`.
- **Classname:** This name is by default the Name and the word `Dialog`, for example `QuestionDialog`. This name will be used during the code generation.
- **Derived From:** The IBM ILOG Views class from which the dialog is derived. Select from the list.
- **Resource Key:** The name of the resource used to reread this file, by default the `.ilv` file name.
- **ILV Filename:** The full path name of the `.ilv` file that was loaded.
- **Title:** The title that appears in the Windows title bar. By default it is the Classname.

- ◆ *Subclass* tab contains the name of the dialog's subclass. This entry is optional.

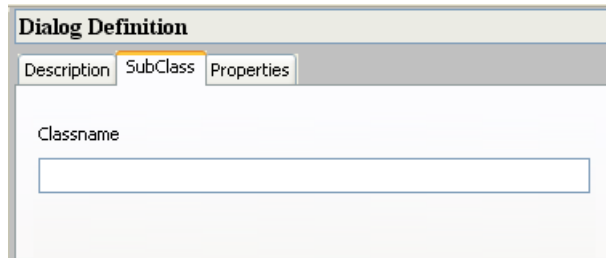


Figure 20.20 'SubClass' Tab of Dialog Definition

- ◆ *Properties* tab allows you to specify standard properties of the dialog box. Choose any or all of these properties.

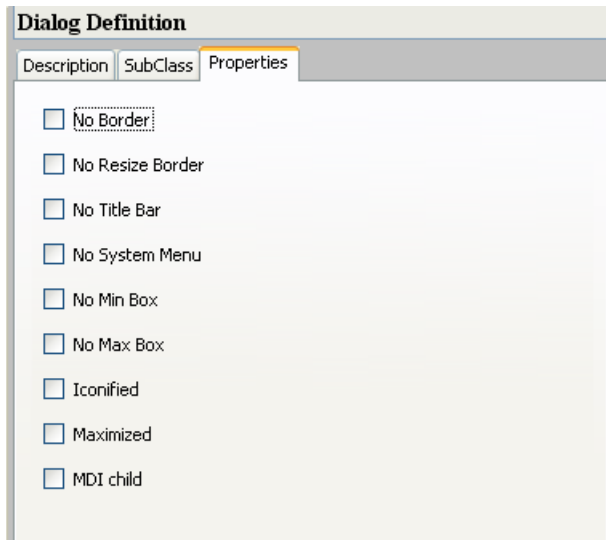



Figure 20.21 'Properties' Tab of Dialog Definition

Creating a Dialog Box

To begin a definition, click the New Dialog button . You are requested to open the predefined `.ilv` file.

Note: To create a dialog box in the Application Framework Editor, you must define a dialog box in IBM ILOG Views Studio and save it (`.ilv` file). This name is requested when creating a dialog box in the Application Framework Editor.

The middle column shows the Dialog tree, with the Dialog Definition on the right. Complete the information for the Dialog Definition tabs (for details see *Dialog Definition*).

Setting Data Parameters

The Application Framework Editor is used to set your data parameters when *Data* is selected from the Application Framework Editor Palette.

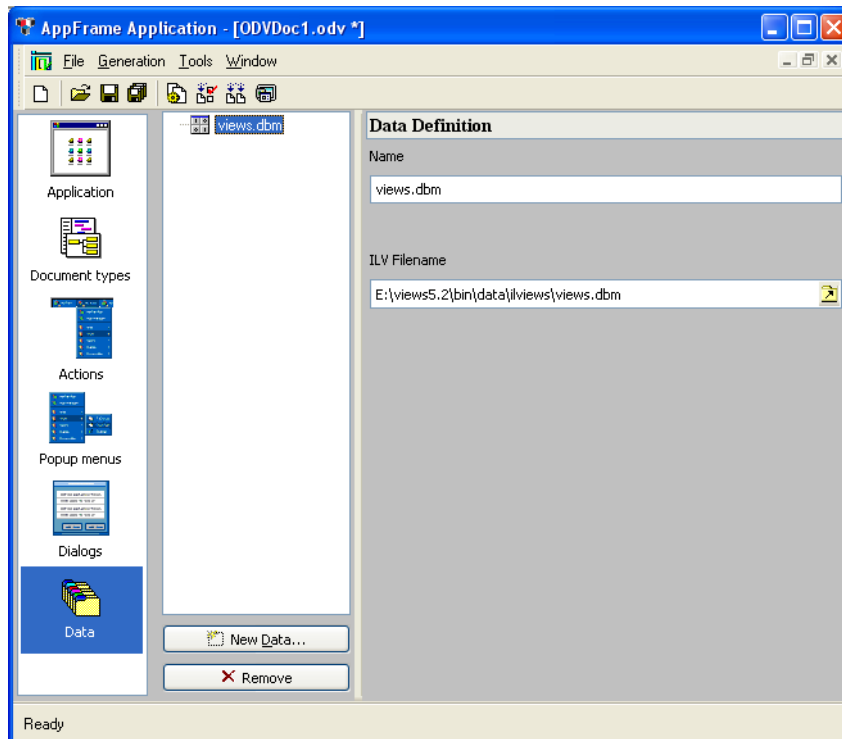




Figure 20.22 'Data' Selected from Palette

You can use this feature to add data files to the application executable. They can be any data files: .dbm, bitmaps, .ilv, or user data files that are not otherwise included.

The Data Definition workspace becomes active when you begin adding a data file by clicking the New Data button .

Data Definition

The Data Definition workspace (see Figure 20.22) allows you to define the data properties needed to include the file.

- ◆ **Name:** The name to be used to reread the file for retrieving the data. The default is the name of the file that was loaded. You can change this name by editing the text field.
- ◆ **ILV Filename:** The full path name of the file. This path name can be changed by clicking  and selecting a new path.

Generating Parameters

After you have defined the application parameters in the Application Framework Editor, you must generate it.

The Generation menu provides commands to generate the application.

Parameters Command

For initial generation, when you select Generation -> Parameters, it displays the Project Generation window.

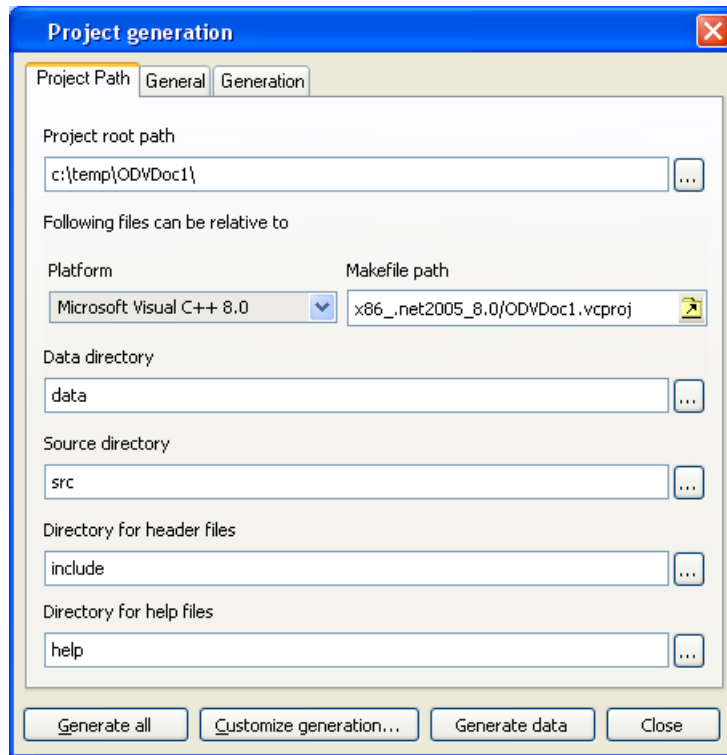


Figure 20.23 Project Generation window, Project Path tab

This window has three tabs:

- ◆ *Project Path* tab (see Figure 20.23) contains the fields:
 - Project Root Path: The root path where the project is saved. The following paths can be relative to this root path.
 - Platform: The platform for the makefile.
 - Makefile Path: The path for the makefile, based on the Platform selection.
 - Directories for: Data, Source, Header files, and Help files. These are all given defaults but can be changed.

- ◆ *General* tab allows you to set general information fields.

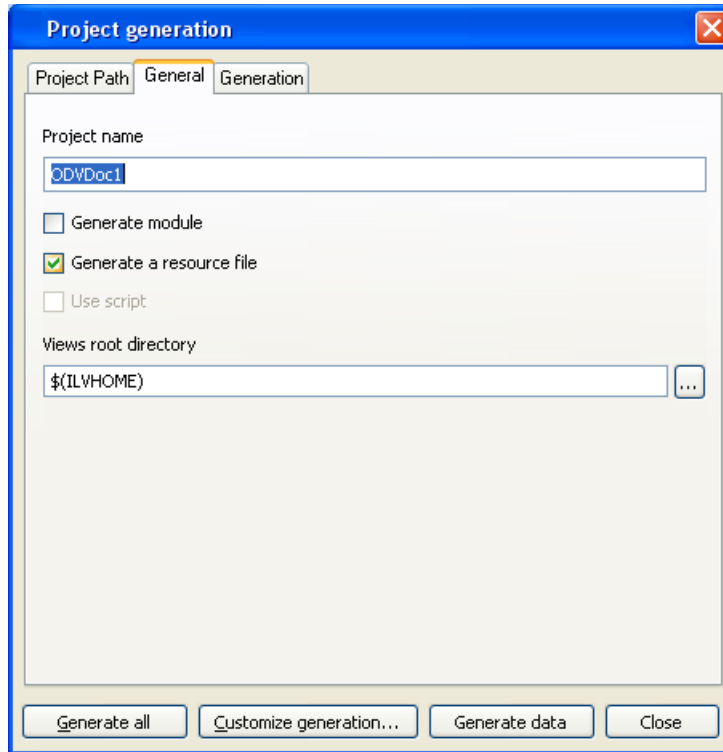


Figure 20.24 'General' tab (Project Generation)

- ◆ *Generation* tab provides information about the project generation.

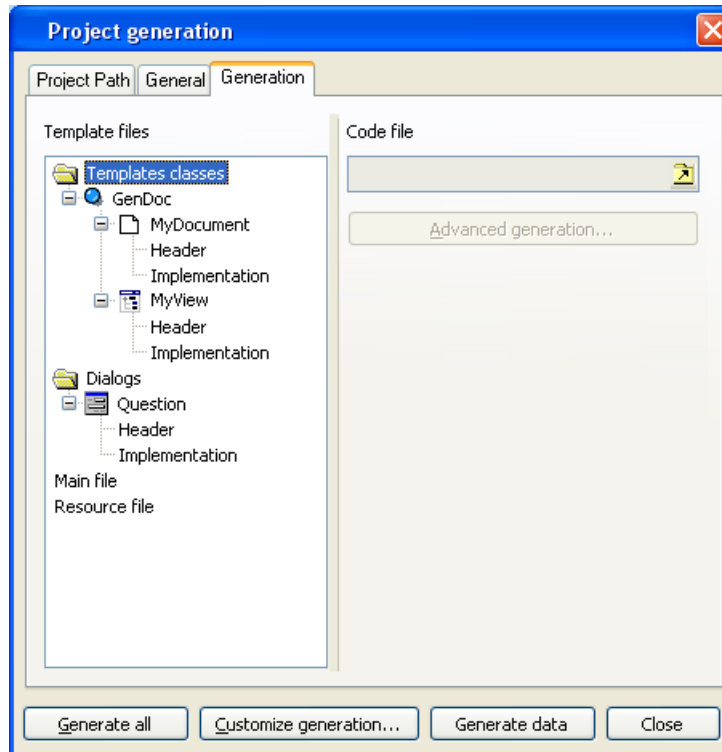


Figure 20.25 'Generation' tab (Project Generation)

Generate All

Use Generate All to generate all files of your application.

Important: This operation replaces all existing generated files of your application. A dialog asks you for confirmation before proceeding.

Custom Generation

Use Custom Generation to generate just one or selected portions of the application. This can be done when adding a new dialog box, for example.

Generate Data

Use Generate Data for updates that do not require changing the source code or makefiles after the initial generation of the application. For example, it can be used to add an action, a document type, a popup, or a new data file.

GUI Action Summary

Table 20.3 Menu and Toolbar Operations











Action	Toolbar Icon	Menu Operation	Comments
File Operations			
Create a new project		File>New	
Load an .odv file		File>Open	
Close the current .odv file		File>Close	
Save an .odv file		File>Save File>Save As	For Save As, type a new name including file extension.
Save all open .odv files		File>Save All	
Generation Operations			
Set project generation parameters		Generation>Parameterst	
Generate the entire application.		Generation>Generate all the application	
Generate specific files in the current application.		Generation>Generate specific files...	Opens the Custom Generation dialog box.
Generate data.		Generation>Generate data	Displays the generation report log.
Tools			
Customize application		Tools>Customize	Opens the Customize window.
Insert and remove modules		Tools>Modules	Opens the Insert/Remove modules dialog box.
Script a project		Tools>Script project	Creates or opens a script project file (.spj).
Window Operations			

Table 20.3 *Menu and Toolbar Operations (Continued)*

Action	Toolbar Icon	Menu Operation	Comments
Start a new window		Window>New window	
Display the next window		Window>Next Window	
Display the previous window		Window>Previous Window	
Cascade all document views		Window>Cascade Windows	
Tile all document views horizontally		Window>Tile Horizontally	
Tile all document views vertically		Window>Tile Vertically	
Quit the Application Framework Editor		File>Exit	Asks about unsaved files before exiting.

Implementing an Application

This chapter discusses how to implement an application under Application Framework, including a description of the classes and files required. It is divided as follows:

- ◆ *How Application Framework Functions*
- ◆ *Option Files*
- ◆ *Main File*
- ◆ *Implementation of a Document Class*
- ◆ *Commands*
- ◆ *Implementation of a Document View Class*

How Application Framework Functions

Application Framework is built on a Document/View architecture (see *The Document/View Architecture*). Figure 21.1 illustrates the different classes that the Document/View mechanism relies upon.

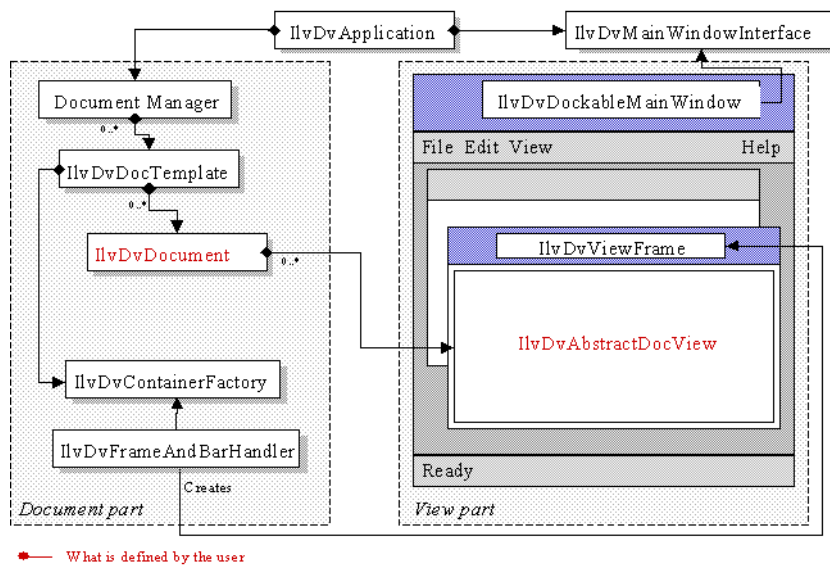


Figure 21.1 Document/View Classes

All the classes shown in the figure, except the `IlvDvApplication` class, the `IlvDvDocument` hierarchy, and the `IlvDvDocViewInterface` hierarchy, are hidden from the developer. Instances of these classes are automatically created according to the application options that are read while the application is initializing.

The code of an Application Framework application consists of:

- ◆ **Option Files:** At least one option file, which is edited using the Application Framework Editor.
- ◆ **Main File:** A main file containing the main entry point of the program, which must instantiate an `IlvDvApplication` (or a derived class) object. This file is generated by the Application Framework Editor and must only be completed in very specific cases, as shown in the `Text` sample.
- ◆ **Implementation of a Document Class:** Files implementing a document class, which is a subclass of `IlvDvDocument`.
- ◆ **Implementation of a Document View Class:** Files implementing a document view class, which is a subclass of `IlvDvDocViewInterface` class.

Option Files

While initializing, an Application Framework application reads three option files that contain data. This data can be the contents of menus and toolbars, the recently used file list, document templates, and so on.

The option files are the following:

- ◆ The Application Framework option file. Its path is `<ILVHOME>/data/ilviews/appframe/docview.odv` and it is contained by the `<ILVHOME>/data/res/appframe.rc` file.

This file contains the descriptions of the default actions (OpenDocument, SaveDocument, Cut, Copy, and so on), the default menus, and the description of the default toolbars.

- ◆ The application option file. The file path is given to the `IlvDvApplication` object using the `IlvDvApplication::setAppOptionsFilename` method.

This file is edited using the Application Framework Editor and contains the following information:

- Application name and title of the main window.
- Description of different document templates.
- The main menu and the toolbars, if different from the default ones stored in the Application Framework option file.
- Description of actions.
- User application data.
- ◆ User profile options file. Its default path is given as follows:

- Windows:

```
<Windows directory>/Profiles/<Username>/Application Data/  
<Application name>.odv
```

- UNIX:

```
$(HOME)/<Application name>.odv
```

To specify a different path, use the method:

```
IlvDvApplication::setUserOptionsFilename
```

This file contains the application data modified by the user the last time the application was run. It is mainly composed of:

- Most Recently Used file list.
- Position and size of the application main window.

- Positions and state (hidden or not) of all dockable toolbars.
- Customizing of toolbar contents.
- Customizing of actions.

Main File

When you create an Application Framework application, you must first create an `IlvDvApplication` object in the main procedure, the same way you create an `IlvApplication` object in a simple IBM® ILOG® Views application:

Note: *The main file is automatically generated using the Application Framework Editor.*

```
int
main(int argc, char* argv[])
{
    IlvDvApplication* app = new IlvDvApplication("", 0, argc, argv);
    IlvDisplay* display = app->getDisplay();
    if (!display || display->isBad()) {
        IlvFatalError("Couldn't create display");
        delete display;
        return -1;
    }
    // Adding the options file
    app ->setAppOptionsFilename((const char*)"myapp.odv");

    // Adding the data base file
    display->getDatabase()->read((const char*)"myapp.dbm", display);

    // Continue...
    application->run();
    return 0;
}
```

`IlvDvApplication` is a subclass of `IlvApplication` and features management of options data and the handling of menu and toolbar items, as well as actions and their states.

Most of all, this `IlvDvApplication` object is aware of all objects involved in the Document/View mechanism (see Figure 21.1). Similarly, all these objects are aware of the application object. The application object is useful, for example, when changing the state of an action from a document or from a document view.

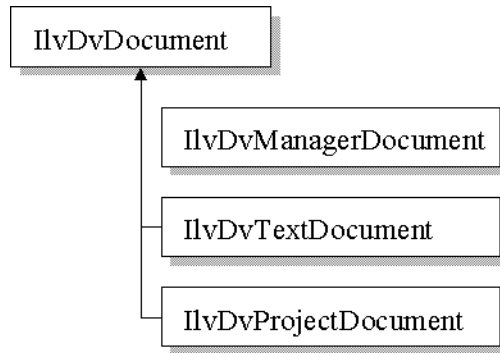
Implementation of a Document Class

A document class is derived from the `IlvDvDocument` class.

The document is *the user data*. The document class loads and saves data and also provides accessors that are used by document views to modify data.

Note: *The user data is similar to the Model View Controller (MVC) approach.*

Application Framework provides document classes that manage specific data, such as IBM ILOG Views managers, text buffers, or projects, as shown in the following inheritance tree:



New Document

A derived document class must override the `IlvDvDocument::initializeDocument` method.

It is called when the File > New command is executed to create a document.

The method must first call `IlvDvDocument::initializeDocument`. Then, it must initialize specific data.

Serialization

The `IlvDvDocument::serialize` method:

```
void IlvDvDocument::serialize(IlvDvStream& stream);
```

is called when a file is opened to create the document, if a call to `stream.isSaving()` returns false. Otherwise, the document must be saved.

Typically, the body of the method has the following form:

```
IlvDvDocument::serialize(stream);
if (stream.isSaving()) {
    // Here, write your persistent data
}
else {
```

```

    // Here, read data from stream
}

```

There are two ways of loading and saving data when using the parameter called stream.

One way is to use `istream` or `ostream` objects. These objects are given by a call to the `istream* getInStream() const` and `ostream* getOutStream() const` methods directly.

The other way, which is usually easier, is to use specific serialization methods, provided by the `IlvDvStream` class. Here are these methods:

◆ Operators

```

// Storing operators
IlvDvStream& operator<<(IlInt i);
IlvDvStream& operator<<(IlUShort w);
IlvDvStream& operator<<(IlShort ch);
IlvDvStream& operator<<(IlUInt u);
IlvDvStream& operator<<(IlBoolean b);
IlvDvStream& operator<<(IlFloat f);
IlvDvStream& operator<<(IlDouble d);
IlvDvStream& operator<<(const IlString& s); // 's' must not contain blanks

// Reading operators
IlvDvStream& operator>>(IlInt& i);
IlvDvStream& operator>>(IlUShort& w);
IlvDvStream& operator>>(IlShort& ch);
IlvDvStream& operator>>(IlUInt& u);
IlvDvStream& operator>>(IlBoolean& b);
IlvDvStream& operator>>(IlFloat& f);
IlvDvStream& operator>>(IlDouble& d);
IlvDvStream& operator>>(IlString& s);

```

◆ `void serialize(IlvString&, IlBoolean betweenQuotes = IlTrue);`

This method is a safe way of loading and saving strings. If the `betweenQuotes` parameter is set to true, the string is saved between quotation marks. This way it can contain blank spaces.

◆ `void serializeBitmap(IlvBitmap*&, IlBoolean lock = IlTrue);`

Serializes a bitmap path.

◆ Serialization of objects

When implementing user classes, it is recommended to derive from the `IlvDvSerializable` class. This class is an abstract interface that provides both a mechanism for safe downcasting and a serialization method:

```
virtual void serialize(IlvDvStream& stream);
```

- `void serializeObjects(IlvArray&);`

Load and save an array of `IlvDvSerializable` objects:

- `void writeObject(const IlvDvSerializable*);`

- `IlvDvSerializable* readObject();`
- ◆ `virtual void clean();`

This method is called to clean up the document data. It is only used for documents whose corresponding document type does not allow opening more than one document (SDI document types, typically project documents). When a user tries to create a new document while a document of the same type is already open, Application Framework does not delete the currently opened document to create another one. Instead, it cleans the open document (by calling this method) and reinitializes the document.

Commands

To modify the data of a document, it is recommended to use the Application Framework command mechanism, which provides the following advantages:

- ◆ Undo/Redo mechanism - The Undo, Redo, and Repeat actions are automatically processed, as well as their state.
- ◆ The modification state of a document is automatically managed. Adding a command to an unmodified document will mark the document as modified (a star will appear in the title of the frames that contain views associated with this document). Similarly, undoing this command will restore the unmodified state of the document (and will remove the star from the title of the same frames).
- ◆ Keeps a log of all modifications made to the document.

Consider the following document class:

```
class MyDocument
: public IlvDvDocument
{
...
    void setX(int x) { _x = x; }
    int getX() const { return _x; }
protected:
    int _x;
};
```

To modify the X property of the document while processing either a document view event/action or a document action, it is not recommended to call directly the `setX` method of the document. It is more appropriate to implement a command class (called `ChangeXPropertyCommand` in this example) that will modify this property:

```
class ChangeXPropertyCommand
: public IlvDvCommand
{
    ChangeXPropertyCommand(MyDocument* document, int newX)
        : _document (document),
          _newX (newX)
    {
```

```

        _oldX = document->getX();
    }
    virtual void doIt() { setX(_newX); }
    virtual void undo() { setX(_oldX); }
    void setX(int x) { _document->setX(x); }
protected:
    MyDocument* _document;
    int _newX;
    int _oldX;
};

```

Therefore, the implementation of a view or the document itself should invoke the following code to change the X property (instead of calling directly the `setX` method of the document):

```
document->doCommand(new ChangeXPropertyCommand(document, newX));
```

This code will execute the `ChangeXPropertyCommand` command by calling its `doIt` method, and will store it within a command history internally managed by the document.

Use the following method of the `IlvDvDocument` class to manage commands:

```
void doCommand(IlvDvCommand* cmd,
              IlBoolean updateUI = IlTrue,
              IlBoolean bSetModified = IlTrue);
```

This method is called to add the command object `cmd` to the history of internal commands. Then, the command is executed by calling its `IlvDvCommand::doIt` method. The `updateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated. The `bSetModified` parameter specifies whether the modification flag of the document must be set to true.

Undo / Redo / Repeat Actions

The Undo, Redo, and Repeat actions are automatically managed by the document. To process these actions, the document invokes the following methods (which can be overridden for specific uses):

- ◆ `virtual IlBoolean canUndo() const;`

This method returns true if the command that has just been executed can be undone. If there is no command that can be undone, for example if the document has not been modified, this method returns false.

- ◆ `virtual void undo(IlBoolean bUpdateUI = IlTrue);`

This method calls the `undo` method of the last command executed. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

- ◆ `virtual IlBoolean canRedo() const;`

This method returns true if the command that has just been undone can be redone by calling the `IlvDvCommand::doIt` method. If there is no command that can be done, for example if the document has not been modified, this method returns false.

- ◆ `virtual void redo(IlBoolean bUpdateUI = IlTrue);`

This method calls the `doIt` method of the last undone command. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

- ◆ `virtual void repeat(IlBoolean bUpdateUI = IlTrue);`

This method repeats the last command executed. This command is copied by calling its `IlvDvCommand::copy` method. The copy is added to the commands history and then executed. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

Reflecting Changes Made In the Data to Associated Views

You have seen how to modify the data of a document by using commands. However, you still need to see how to notify the views associated with the document to reflect these changes.

A document can have several views of different types. Therefore, to communicate with its views, a document sends generic messages that are interpreted by each view depending on their type. To send generic messages to its views, a document uses the following method:

```
void notifyViews(const char* messageName,
                IlvDvDocViewInterface* exceptView, ...);
```

- ◆ The name of the message is `messageName`. It must not be the name of an action (such as Copy, OpenDocument, and so on).
- ◆ The `exceptView` parameter specifies a view that must not be notified. The value of this parameter is usually 0. It can also be the view returned by the call to `getCurrentCallerView()` (this method returns the view that is currently notifying the document of an event). In this case, you may want this view *not* to be notified because it may have already modified its contents before it notified the document.
- ◆ The variable list of parameters that follows depends on the message name. For example, if a document contains information on an employee and that a command has just changed the name of that employee, the document notifies its views of this change as follows:

```
void EmployeeDocument::changeName(const char* name)
{
    _employeeName = name;
    notifyViews("NameChanged", 0 /* Notify all views */, name);
}
```


To receive a message, a view (or any other class that implements the `IlvDvInterface` interface) must specify an entry in its interface declaration. This entry makes a message name and its parameters correspond with a method of the class. This is explained in more detail in Chapter 22, *Application Framework Interfaces*.

The sample can now be completed so that the view of an employee document can receive the message `NameChanged`:

```
IlvDvBeginInterface(EmployeeView)
/* The message "NameChanged" with one parameter const char* name
   is processed by the method:
   EmployeeView::nameChanged with one parameter const char* name */

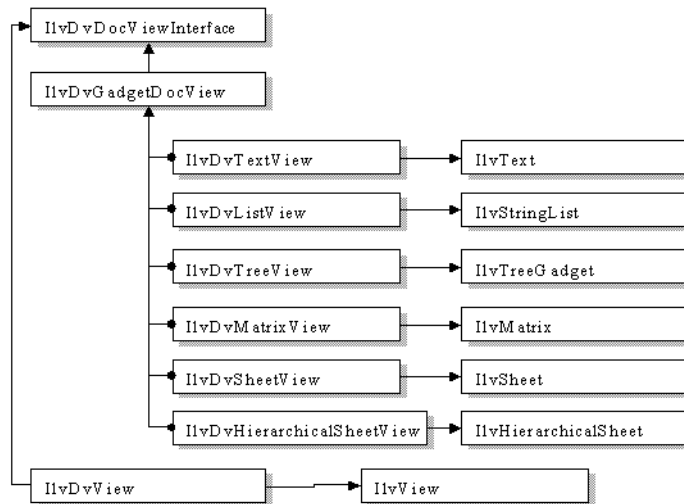
   Method1(NameChanged, nameChanged, const char*, name)
IlvDvEndInterface1 (IlvDvFormView)

/* The nameChanged method is automatically called when an EmployeeDocument
   notifies its views giving the message name "NameChanged" */

void EmployeeView::nameChanged(const char* name)
{
    IlvTextField* nameField = getEmployeeNameField();
    nameField->setLabel(name, IlvTrue);
}
```

Implementation of a Document View Class

A document view class is derived from the `IlvDvDocViewInterface` class. It shows the contents of its associated document and allows the end user to edit it. Here is the inheritance tree of the `IlvDvDocViewInterface` class:



When deriving from this class, only the `IlvDvDocViewInterface::initializeView` method needs to be overridden.

```
virtual void initializeView();
```

This method is called to initialize the document view object according to the document data. For example, a list view can be filled according to a data set stored in the document. A sample of the body of this method is shown here:

```
void
ListView::initializeView()
{
    IlvDvListView::initializeView();
    ListDocument* document = getListDocument();
    IlvUInt count;
    Element* const* elements = document->getElements(count);
    for(IlvUInt iElement = 0; iElement < count; iElement++)
        addString(elements[iElement]->getName());
}
```

Interactions

You have seen how a document view can show the contents of its document.

However, you will also need to edit the contents of a document by interacting with a view. You will want interactions made on the view to be translated into changes in the document data.

Reminder: *It is recommended to use Application Framework commands to do this.*

Consider a list view that lets the user edit a list of names stored within the document associated with the view. You want the user to be able to remove a name (the selected name in the view) by pressing the Del key. To do so, proceed as follows:

```
// The list view tracks the event and makes the changes to the document
// through a command
void ListView::handleGadgetEvent(IlvEvent& event)
{
    if (event.type() == IlvKeyDown) {
        IlvUShort c = event.data();
        if (c == IlvDeleteKey) {
            getNamesDocument()->doCommand(new RemoveNameCommand(getNamesDocument(),
                                                                    getSelectedName());
            return IlvTrue;
        }
    }
    return IlvStringList::handleGadgetEvent(event);
}

// Here is the implementation of the command class
class RemoveNameCommand
: public IlvDvCommand
{
public:
    RemoveNameCommand(NamesDocument* document, const char* name):
        _document(document), _name(name) {}
    virtual void doIt() { _document->removeName((const char*)_name); }
    virtual void undo() { _document->insertName((const char*)_name); }
protected:
    NamesDocument* _document;
    IlString _name;
};
```

This showed how events that occur on a view can be reflected to a view through the use of commands. However, the view still has to be refreshed to reflect this change.

In this sample, the selected name item still has to be removed from the list when the user presses the Del key. To do this, the document notifies its associated views when it removes a name from its list of names. To communicate with its views, the document sends generic messages to its associated views, as shown in section Commands.

The sample will now be completed:

```
// The document notifies its views when it removes a name from its
// list of names
void NamesDocument::removeName(const char* name)
{
    _namesArray.removeName(name);
    notifyViews("RemoveName", 0, name);
}

// The list view updates its list when the document notifies it that it
// has removed a name from its list. First, the list view class associates
// the RemoveName message with its removeName method. Thus, this method will
// be called when the user notifies its views with the RemoveName message.
```

```
IlvDvBeginInterface(ListView)
Method1(RemoveName, removeName, const char*, name);
IlvDvEndInterface1(IlvDvListView)
void ListView::removeName(const char* name)
{
    IlShort index = getPosition(name);
    if (index != (IlShort)-1) {
        remove(index);
        redraw();
    }
}
```

For more information on managing events in document views, see `samples manager` and `synedit`, both provided in the `samples` directory.

Application Framework Interfaces

This chapter describes how to use the Application Framework interface. It is divided as follows:

- ◆ *The Interface Mechanism*
- ◆ *Declaring an Interface*
- ◆ *Naming Convention for Macros*

The Interface Mechanism

Application Framework provides an interface mechanism that allows you to:

- ◆ Track and process GUI actions (see the chapter on *Actions*).
- ◆ Perform introspection on your classes.
- ◆ Script your classes.

This interface mechanism associates a name with a method or field of a class. The name of this method or field depends on how the interface mechanism is used. For introspection and scripting, the name is a key that identifies the method.

Declaring an Interface

Here is a small sample showing how to declare an interface to a class (called 'A'):

```
// Declaration of class A
class A
: public IlvDvInterface
{
public:
    void setX(int x) { _x = x; } < /FONT >
    int getX() const { return _x; }

protected:
    int _x;
};

// Implementation file of A. Use the following macros to
// introspect methods setX, getX, and field _x:

IlvDvBeginInterface(A)
    Method1(SetX, setX)
    TypedMethod (GetX, getX)
    Field(X, _x)
IlvDvEndInterface()

....
// Using an instance of A as an interface, it is possible
// to invoke its methods and to modify its field
// without being aware of class A !!!
A* a = new A;
IlvDvInterface* interf = a;

// First we invoke its methods:
IlvDvValue returnedValue;
interf->callMethod(IlvGetSymbol("SetX"), &returnedValue, 100);
interf->callMethod(IlvGetSymbol("GetX"), &returnedValue);
assert((IlvInt) returnedValue == 100);

// Then, we modify its field directly:
interf->setFieldValue(IlvGetSymbol("X"), 200);
assert((IlvInt)interf->getFieldValue(IlvGetSymbol("X"),
                                   &returnedValue) == 200);
```

Naming Convention for Macros

This section discusses the naming conventions for macros used for scripting and introspection.

For methods:

- ◆ The root of the macro name must be Method.

- ◆ If the declared method returns a value, the macro must begin with the prefix `Typed`.
- ◆ If the declared method contains arguments, the macro must end with the suffix `[Number of Parameters]`.

For fields:

- ◆ The macro name is `Field`.

Examples are included in Table 22.1:

Table 22.1 *Macro samples*

Methods to “export”	Macro declarations
<code>Violate getPosition() const;</code>	<code>TypedMethod (GetPosition, getPosition, llvFloat)</code>
<code>const char* set(int);</code>	<code>TypedMethod1 (Set, set, int, ExportedFirstParameterName, const char*)</code>
<code>void setPosition(int x, int y);</code>	<code>Method2 (SetPosition, setPosition, int, X, int, Y)</code>

For scripting and introspection, the first macro parameter is used to identify the method or the field given as the second parameter.

For more information on introspection, see the sample dealing with introspection provided in the *samples* directory.

Actions

This chapter describes how to use the action events provided with Application Framework. It has the sections:

- ◆ *Activating an Action Event*
- ◆ *Processing an Action Event*

Activating an Action Event

Using interfaces, Application Framework provides a mechanism that makes it easy to process actions. The application processes the activation of a menu item in a menu or in a toolbar by generating an action event. This action event is generated according to the action associated with the activated menu item.

Then, the action event is sent to the following targets in this order:

- ◆ The active document view, which is the active view inside the active view frame.
- ◆ The document associated with the active document view.
- ◆ The active view frame.
- ◆ The views and their associated documents, which are inserted into dockable bars.
- ◆ The main window.

- ◆ The action processors declared to the application.
- ◆ The application itself.

Processing an Action Event

To process an action event, a class must insert an `Action` macro in its interface. The first parameter of the `Action` macro is the name of the action, and the second parameter is the name of the method that will process this action.

For example, here is a text view that manages the `Cut` action event:

```
IlvDvBeginInterface(MyTextView)
    Action(Cut, myCut)
IlvDvEndInterface1(IlvDvTextView)

void
MyTextView::myCut()
{
    ...
}
```

A document or a view can manage the action state the same way as it processes an action event. It does this using the macro `RefreshAction([ActionName], [MethodName])`.

For example, here is a text view that manages the `Cut` action state:

```
IlvDvBeginInterface(MyTextView)
    RefreshAction(Cut, refreshCut)
IlvDvEndInterface1(IlvDvTextView)

void
MyTextView::refreshCut(IlvDvActionDescriptor* desc)
{
    desc->setValid(isRelevantSelection());
}
```

The `refreshCut` method will be called each time the document (and its associated document views) becomes active. Since this may not be sufficient, it is possible to force the checking of the action state by calling the application method `refreshAction([ActionName])`. In the previous sample, `refreshAction(IlvGetSymbol("Cut"))` can be invoked, for example, each time the selection changes in the text view.

Editing States

States let you predefine different contexts, or states, for your application. In a particular context, your application may open or close panels, hide or show graphic objects, change their sensitivity, colors or any other properties. All these settings are called *state requirements*. A state is just a set of state requirements. As a general rule, it is not recommended to modify these settings through programming when they belong to contexts that are handled by the state mechanism.

IBM® ILOG® Views Studio lets you interactively define states and their requirements for your application.

This appendix provides an example of how to use the state mechanism of IBM ILOG Views Studio. It is divided into two sections:

- ◆ *Creating a Simple Application*
- ◆ *Editing the Show State*

Creating a Simple Application

In the following example, you will create a simple application with two panels. Only one of the panels is visible when the application is started. The objective is to open the second panel by clicking a button on the first panel.

A. Editing States

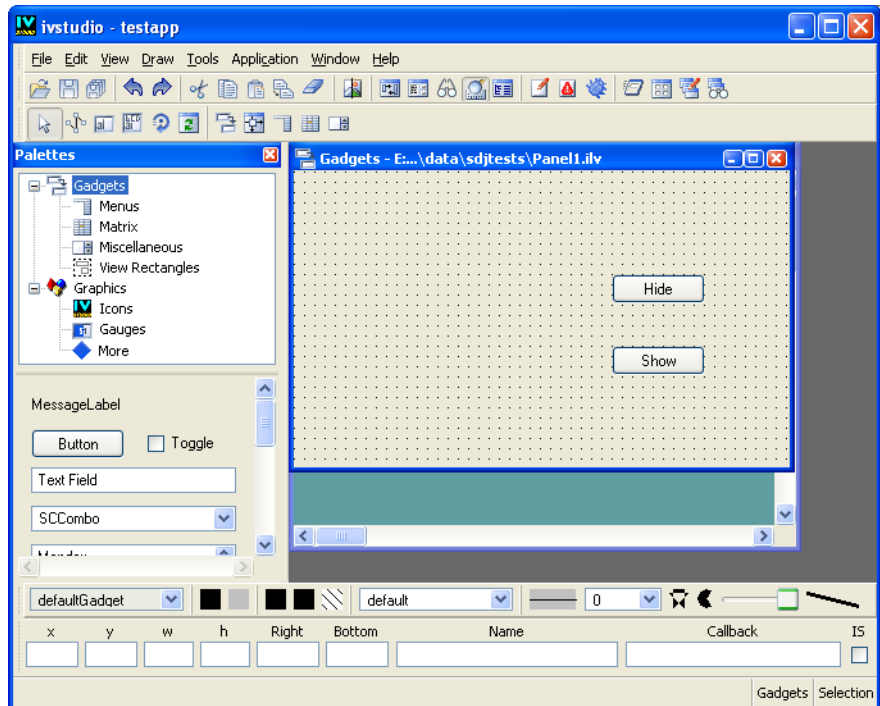
If you are already working in IBM® ILOG® Views Studio, start by choosing Close All from the Window menu to close all open buffers. Then, open a new application buffer window by choosing New from the File menu and Application in the submenu that appears.

Creating the First Panel

To create the first panel, do the following:

1. Choose New from the File menu and then choose Gadgets in the submenu to open a new Gadgets window buffer.
2. Click Gadgets in the tree in the upper pane of the Palettes panel.
3. Drag two buttons to the Gadgets buffer window from the lower pane of the Palettes panel.
4. Double-click the buttons to open the associated inspector panel.
5. In the Name field of the General page, type `ShowButton` and `HideButton`.
6. In the Label field of the Specific page, type `Show` and `Hide`.
7. Resize the panel so it has a suitable size.
8. Save the Gadgets buffer window as `panel1.ilv` in a directory of your choice.

The Panel1 should have the following appearance:



9. Click the Panel Class Palette icon in the Main window toolbar to open the Panel Class palette.
10. Click the New Panel Class icon in the Panel Class palette to create the Panel1 panel class.
To do so, make sure that the `panel1.ilv` Gadgets buffer window is activated.
11. Click the Application buffer window to bring it to the foreground.
You can also choose `<Application>` from the Window menu.
12. Drag the Panel1 icon from the Panel Class palette and drop it in the Application window buffer.

Creating the Second Panel

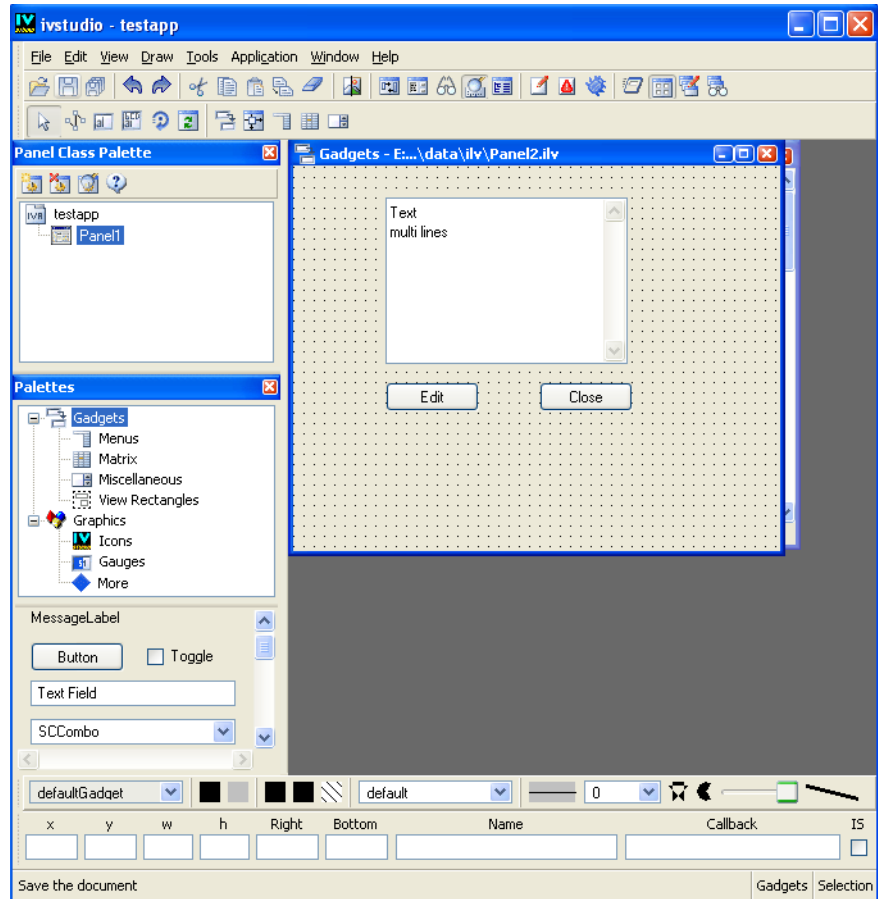
You are now going to create a second panel.

1. Choose New from the File menu and then Gadgets in the submenu to open a new Gadgets buffer window.
2. If necessary, click Gadgets in the tree in the upper pane of the Palettes panel.

A. Editing States

3. Drag a multiline text gadget (`IlvText`) to the current buffer window.
4. Double-click the text gadget to open its inspector panel.
5. In the Name field of the General page, type `Text`.
6. Drag two buttons below that text.
7. Double-click the buttons to open the associated inspector panel.
8. In the Name field of the General page, type `EditButton` and `CloseButton`.
9. In the Label field of the Specific page, type `Edit` and `Close`.
10. Save the buffer as `panel2.ilv` in the same directory.

Panel2 should have the following appearance:



11. Click the New Panel Class icon in the Panel Class palette to create the Panel2 panel class.

To do so, make sure that the `panel2.ilv` Gadgets buffer window is activated.

12. Click the Application buffer window to bring it to the foreground.

You can also choose <Application> from the Window menu.

13. Drag the Panel2 icon from the Panel Class palette and drop it in the Application window buffer.

14. Double-click the title bar of Panel2 to open its inspector.

15. Turn off the Visible toggle button in the General page of the Panel Instance inspector.

16. Click the Test button in the Main window toolbar.

You can see that Panel2 is not visible at application start-up.

Click the Test button again to close the test panel.

17. Save the application as `myapp.iva` in the same directory.

States Panels

IBM® ILOG® Views Studio provides you with two separate panels to edit states:

- ◆ The State Tree panel for managing the whole state hierarchy of the application.

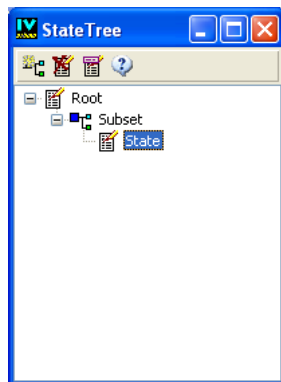


Figure A.1 The State Tree Panel

- ◆ The State inspector panel for inspecting the properties of the state selected in the State Tree.

A. Editing States

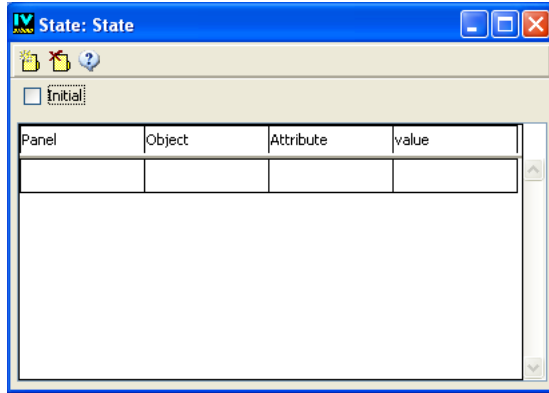


Figure A.2 The State Inspector Panel

To open these panels, you can use the Commands panel of IBM ILOG Views Studio. Click the Commands icon in the Main window toolbar. Then select EditStates from the list of commands in the Commands panel.

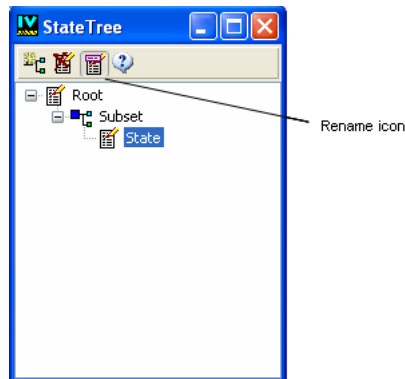
If your application has no defined state, the Edit State command creates a root state, a subset, and a state. The state subsets will be discussed later in this chapter.

Editing the Show State

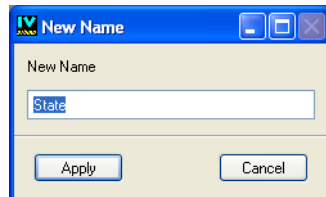
You need two states for the application. The first one is the initial root state where only the first panel is visible. The second is a state where Panel2 is visible.

To name this second state Show:

1. Select State in the State Tree panel.
2. Click Rename in the State Tree toolbar.



A dialog box opens allowing you to enter a new name for the selected state.



3. Type Show and click Apply.

The application now has two states: Root and Show. Using the State inspector, you can define the requirements for each state. You want Panel2 to be visible when the application is in the Show state.

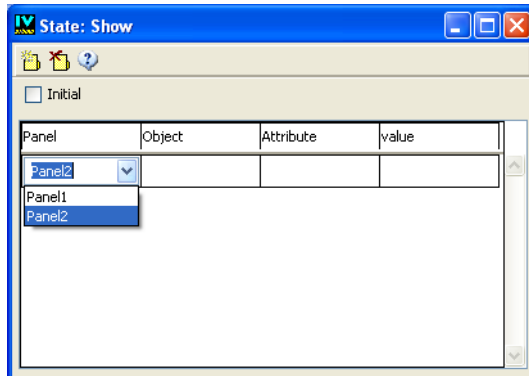
To do this, you are going to set the visible attribute for Panel2 to true:

1. If necessary, activate the Application buffer window.
2. Make sure the Show state is selected in the State Tree panel.
3. In the State inspector, click in the first row of the Panel column.

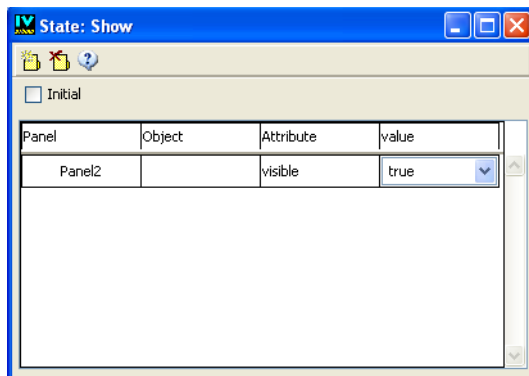
A combo box appears with a list of the panel instances from the application.

4. Select Panel2 from the list.

A. Editing States



5. Click in the Attribute column.
A list of state requirements related to the panel is displayed.
6. Choose `visible` from the list.
7. Click in the Value column.
A list of values related to `visible` is displayed.
8. Choose `true` from the list.



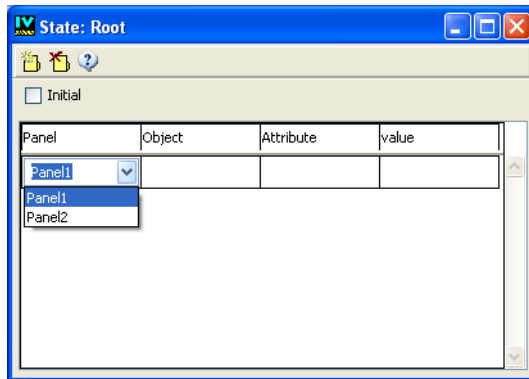
The target of this state requirement is a panel. It is identified by the panel name. Since the target of the state requirement is not an object, the Object column remains blank.

Chaining States

When the application is launched, the Root state is automatically selected. You want to be able to go to the Show state by clicking the Show button.

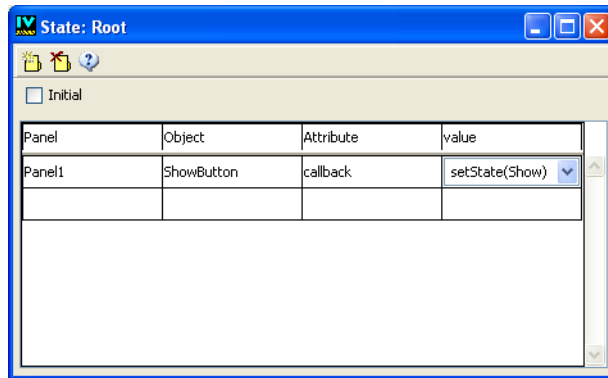
To attach the Show state to the Show button, follow these steps:

1. If necessary, activate the Application buffer window.
2. Select the Root state in the State Tree panel.
3. In the State inspector, click in the first row of the Panel column.
A combo box appears with a list of the panel instances from the application.
4. Select Panel1 from the list.



5. In the Object column, select ShowButton from the list of objects in the combo box.
6. Click the Attribute column.
A list of state requirements related to `I1vButton` objects is displayed.
7. Choose `callback` from the list.
8. Click the Value column of the sheet.
A list of related callbacks is displayed.
9. Choose `setState(Show)`.

A. Editing States



When you test your application, only Panel1 is visible. But now, when you click the Show button, Panel2 is displayed and you are in the Show state.

Changing the Label and the Callback of the Show Button

The states mechanism provides you with predefined callbacks that let you set or leave a state. A callback is a state requirement attached to an object in a particular state. It can be overridden in different states.

In the next exercise, you want the Show button to bring the application from the Show state back to the Root state. You also want to change the button label so it displays Root when the Show state is selected.

1. Select the Show state in the State Tree panel.
2. If necessary, click the New requirements icon at the top-left of the State inspector to add a new row.

3. In the State inspector, click in the empty row of the Panel column.

A combo box appears with a list of the panel instances from the application.

4. Select Panel1 from the list.
5. In the Object column, select ShowButton from the list of objects in the combo box.
6. Click in the Attribute column and choose `label` from the list.

Scroll down the list to make `label` appear.

7. Type `Root` in the Value field and press Enter.

A new row is automatically added when you reach the last column and press Enter.

8. Click in the Attribute column of the new empty row.

The panel and object names are automatically copied from the previous line and the related state requirement names are displayed.

9. Choose `callback`.
10. Click in the Value column and choose `leaveState(Show)`.

When you test your application and go into the Show state (by clicking the Show button), the label of the Show button changes to Root. When you click it once again, you are back to the initial Root state. The label and callback of the Show button are restored and Panel2 is hidden. In short, when you leave a state, the properties modified by the state requirements are restored.

Creating a Substate: the Edit State

You do not want the text field in Panel2 to be editable when this panel is in the Show state. You do, however, want to be able to edit the text in the Edit state.

To make the text field in Panel2 noneditable when the Show state is active, do the following:

1. Select the Show state in the State Tree panel.
2. If required, click the New requirement icon in the State Inspector to create a new row.
3. In the Panel column of the new row, select Panel2 from the list of panels in the combo box.
4. In the Object column, select Text from the list of objects in the combo box.
5. Click in the Attribute column and choose `editable` from the list.

You will notice that the list of related requirements is not the same as the one you chose for a button, as the requirements depend on the object type.

6. Click in the Value column and choose `false`.
7. Test the application and verify that the Text field is not editable in the Show state.

You are now going to define a substate of Show that will inherit from its visibility requirement: the Edit substate. To do so:

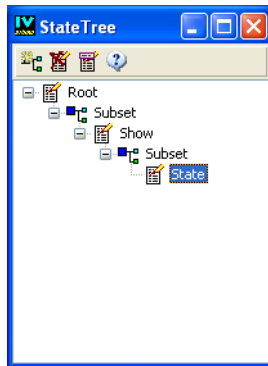
1. Select the Show state in the State Tree panel.
2. Click the New Subset icon in the State Tree toolbar.

A new tree item is created and selected. Notice that the toolbar has slightly changed—the New Subset icon is now replaced with the New State icon.

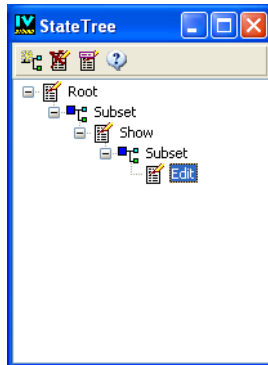
3. Click the New State icon in the toolbar.

A new state item is created in the tree.

A. Editing States

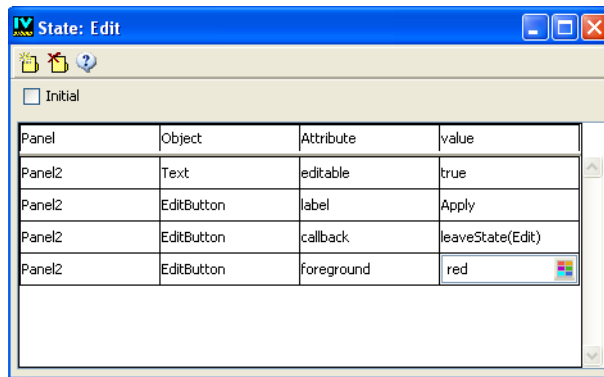


4. Click Rename to rename the state to Edit.



5. Leave the Edit state selected in the State Tree or select it if needed.
6. If required, click the New requirement icon in the State Inspector to create a new row.
7. In the Panel column of the empty row, choose Panel2 from the combo box.
8. In the Object column, choose Text from the list of objects.
9. Choose `editable` in the Attribute column.
10. Choose `true` in the Value column.
11. Click New Requirement in the State Inspector toolbar to create a new row in the State Inspector panel.
12. Choose Panel2 in the Panel column and EditButton from the Object column.
13. Choose `label` in the Attribute column.

14. Type `Apply` in the Value column and press Enter (or click New Requirement in the toolbar).
15. In the empty row just below, click in the Attribute column and select `callback`.
16. Choose `leaveState(Edit)` in the Value column.
17. Create another state requirement for this button and choose `foreground` as its attribute.
18. Click the Value item and select a color from the color selector (for example, red).

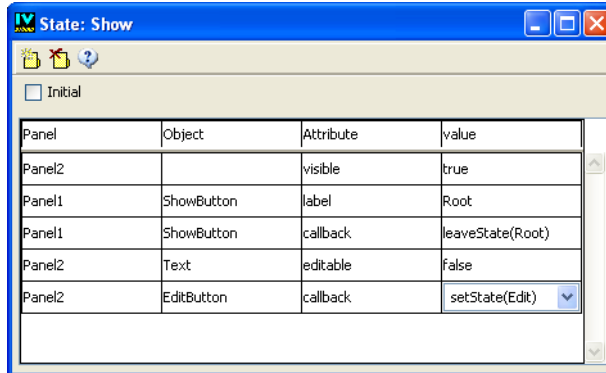


Notice that when you click the Value column in the State Inspector panel, the behavior of the inspector depends on the chosen attribute. It may activate a string list of predefined values or a specialized selector.

Now, go back to the Show state to attach the Edit state to the Edit button in Panel2:

1. Select the Show state in the State Tree panel.
2. In the State inspector, create an empty row.
3. Choose Panel2 from the combo box of the Panel column and EditButton from the combo box of the Object column.
4. Choose `callback` in the Attribute column.
5. Choose `setState(Edit)` in the Value column.

A. Editing States

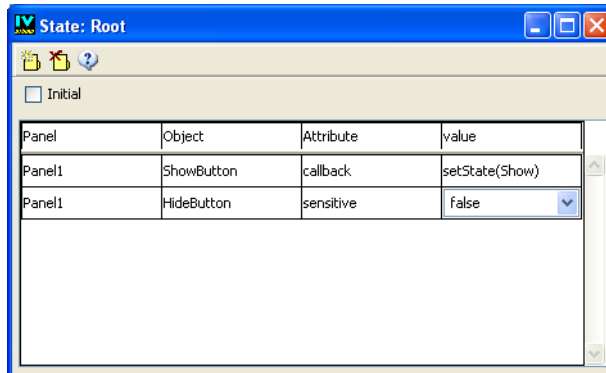


Panel	Object	Attribute	value
Panel2		visible	true
Panel1	ShowButton	label	Root
Panel1	ShowButton	callback	leaveState(Root)
Panel2	Text	editable	false
Panel2	EditButton	callback	setState(Edit)

Initial

- Now, test the application. Once you are in the Show state, go to the Edit state by clicking the Edit button in Panel2.

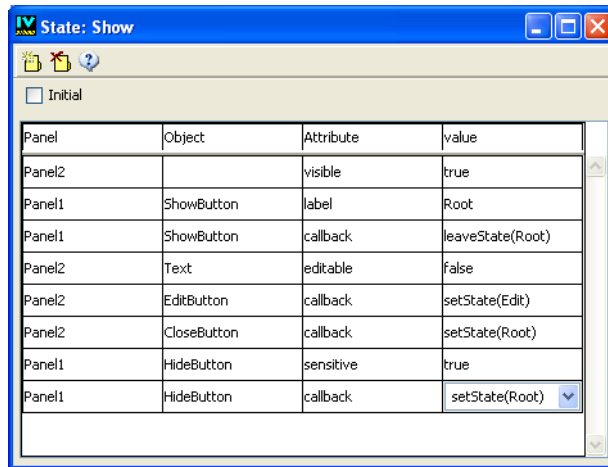
You can also enhance your application states by modifying and adding the state requirements so they look like this:



Panel	Object	Attribute	value
Panel1	ShowButton	callback	setState(Show)
Panel1	HideButton	sensitive	false

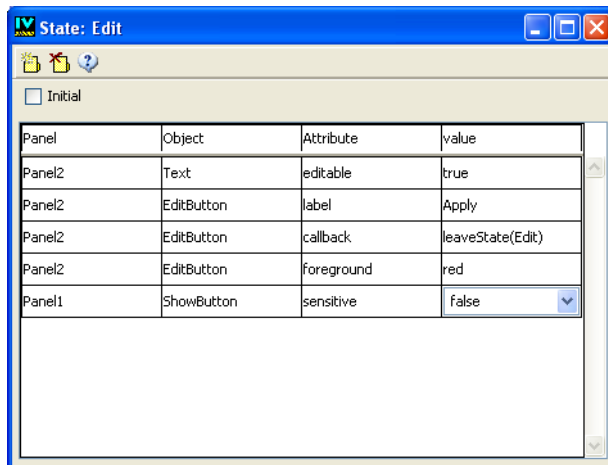
Initial

Figure A.3 The Root Sheet of the State Inspector



Panel	Object	Attribute	value
Panel2		visible	true
Panel1	ShowButton	label	Root
Panel1	ShowButton	callback	leaveState(Root)
Panel2	Text	editable	false
Panel2	EditButton	callback	setState(Edit)
Panel2	CloseButton	callback	setState(Root)
Panel1	HideButton	sensitive	true
Panel1	HideButton	callback	setState(Root)

Figure A.4 The Show Sheet of the State Inspector



Panel	Object	Attribute	value
Panel2	Text	editable	true
Panel2	EditButton	label	Apply
Panel2	EditButton	callback	leaveState(Edit)
Panel2	EditButton	foreground	red
Panel1	ShowButton	sensitive	false

Figure A.5 The Edit Sheet of the State Inspector

The State File

When you save an application that has defined states, the state definitions are saved in a `.ivs` file, with the same name and directory as the application file. This file is automatically loaded when you load the corresponding application file. In the generated C++ code, the state file is loaded if there are defined state requirements. The state file must be found in a directory specified by the `ILVPATH` environment variable (or the equivalent resource).

A. Editing States

Index

Symbols

.odv files **380**
 .spj files **405**

Numerics

2D Graphics buffer window
 description of **27**

A

accessors

- building mode **170**
- combined **172**
- dependent **171, 172**
- panels **127**
- preconditions **179**
- properties **169**
- tree **175**
- update mode **170**
- validators **180**

action

- creating **393**

action event

- processing **427**

action parameters

- setting **389**

action state

- forcing **427**

actions

- description **426**

activate member function

- IlvButton class **234**

ActivateCallbackType member function

- IlvTreeGadget class **276**

activateMatrixItem member function

- IlvMatrix class **314**

activating objects **59**

Active editing mode **59**

Active mode **32**

addChangeLookCallback member function

- IlvDisplay class **222**

addField member function

- IlvSpinbox class **256**

addGuide member function

- IlvGraphicHolder class **207**

adding

- menu item **381**

- popup menu item **396**

- popup submenus **396**

- toolbar item **382**

addItem member function

- IlvTreeGadget class **272, 273**

addLabel member function

- IlvSpinbox class **257**

addLine member function

- IlvText class **263**

addObject member function

- IlvSpinBox class **257**

- addpage member function
 - IlvNotebook class **244**
- addPane member function
 - IlvPanedContainer class **325**
- AddPanel command **40**
- addRelativeDockingPane member function
 - IlvDockableMainWindow class **345**
- adjustLast member function
 - IlvMatrix class **307**
- alignmentBaseClass option **136**
- allowEdit member function
 - IlvMatrix class **313**
- Application buffer window **84, 95**
 - default **28**
 - description of **28**
 - editing **28**
 - opening **88**
 - saving **88**
- application files
 - inserting code **132**
- Application Framework
 - code **409**
 - inheritance tree **412**
 - overview **370, 426**
- Application Framework Editor
 - creating an application **378**
 - description **371**
 - develop an application **378**
 - document type **379**
 - main window **375**
 - menu **405**
 - startup **374**
 - toolbar **377, 405**
 - using **374**
 - workspace **377**
- Application inspector
 - description of **49**
 - General page **90**
 - Options page **91**
- application name **381**
- application parameters
 - setting **380**
- applicationBufferBackground option **137**
- applicationFileExtension option **137**
- applicationHeaderFile option **137**

- applications
 - building an example **112**
 - C++ code generation **121**
 - default **84**
 - derived classes **129**
 - description files **88**
 - editing **84, 95**
 - general properties **90**
 - generating **405**
 - generating code **92**
 - generating data **405**
 - generating specific files **405**
 - header files **93**
 - inspecting **89**
 - opening **375**
 - setting options **91**
 - source files **93, 127**
 - testing **111, 129**
 - viewing properties **89**
- applyResources member function
 - IlvGraphic class **216**
- areLabelsVertical member function
 - IlvNotebook class **243**
- arrow buttons **233**
 - setting the arrow direction **233**
- attach member function
 - IlvGraphicHolder class **207**
- attaching objects **61, 63**
 - setting guides **62**
 - testing **66**
- Attachment editing mode **61**
- Attachments icon **61**
- Attachments inspector panel **63**
- Attachments mode **33**
- autoFitToSize member function
 - IlvMatrix class **307**
- autoLabelAlignment member function
 - IlvStringList class **260**

B

- betweenQuotes **413**
- bitmaps
 - insensitive **234**
 - sensitive **234**

- toggle buttons **269**
- BitmapSymbol member function
 - IlvGadgetItem class **281**
- buffers
 - initializing **149**
- buttons **233**
 - callbacks **234**
 - displaying bitmaps **234**
 - displaying the frame **234**
 - handling events **234**
 - mnemonics **234**

C

- C++
 - prerequisites **20**
- C++ code
 - generating **121**
- callback method
 - generating **124**
- callback types
 - gadgets **210**
- callbacks **123, 124**
 - buttons **234**
 - defining **133**
 - in panel classes **100**
 - string lists **261**
- cascade document views **406**
- cellBBox member function
 - IlvAbstractMatrix class **305**
- cellInfo member function
 - IlvAbstractMatrix class **304**
- changeSelection member function
 - IlvNotebook class **247**
- check member function
 - IlvText class **263**
 - IlvTextField class **266**
- checkLabelOrientation member function
 - IlvAbstractBarPane class **342**
- client views **350**
- closing
 - Application Framework Editor **406**
 - ODV file **405**
- columnBBox member function
 - IlvAbstractMatrix class **305**

- columns member function
 - IlvAbstractMatrix class **303**
- columnSameWidth member function
 - IlvAbstractMatrix class **303**
- combo boxes **235**
 - callbacks **236**
 - editable **235**
 - localizing **236**
 - setting the selected text **236**
- command classes
 - predefined **143**
- command description, initializing **149**
- commands
 - adding **143**
 - errors **143**
- compareItems member function
 - IlvGadgetListItemHolder class **289**
- components palette **376**
- configuration options **136**
 - alignmentBaseClass **136**
 - applicationBufferBackground **137**
 - applicationFileExtension **137**
 - applicationHeaderFile **137**
 - defaultApplicationName **137**
 - defaultCallbackLanguage **137**
 - defaultHeaderDir **137**
 - defaultHeaderFileScope **137**
 - defaultObjDir **137**
 - defaultSrcDir **138**
 - defaultSystemName **138**
 - headerFileExtension **138**
 - JvScriptApplication **138**
 - makeFileExtension **138**
 - noPanelContents **139**
 - panelBaseClass **139**
 - sourceFileExtension **139**
 - system **139**
 - toolbarItem **140**
 - userSubClassPrefix **140**
 - userSubClassSuffix **140**
- createItem member function
 - IlvGadgetItemHolder class **285**
- createSliderPane member function
 - IlvPanedContainer class **328**
- creating

- action **393**
- application **378**
- gadgets **55**
- menu bars **66**
- menu items **381**
- objects **55**
- options file **380**
- panel class instances **102**
- panel classes **97**
- panels **55**
- popup menu **395**
- popup menu items **396**
- pop-up menus **69**
- popup submenu **396**
- toolbar items **382**
- creating objects **55**
- custom generation **404**
- customize tool **405**

D

- data
 - generating **405**
- data files **88, 89**
- data parameters
 - setting **400**
- date fields **237**
 - formats **248**
 - setting a date value **238**
 - setting the format **237**
- dates
 - fields **237**
 - formats **237**
- DecrementCallbackType member function
 - IlvSpinBox class **258**
- default application **84**
- defaultApplicationName option **137**
- defaultCallbackLanguage option **137**
- defaultHeaderDir option **137**
- defaultHeaderFileScope option **137**
- defaultItemsSize member function
 - IlvAbstractBar class **299**
- defaultObjDir option **137**
- defaultSrcDir option **138**
- defaultSystemName option **138**

- derived classes
 - defining **129**
 - using **130**
- description files
 - state **442**
- desktop managers **351**
- desktop views
 - creating **351**
- detachItem member function
 - IlvTreeGadget class **273**
- developing
 - application **378**
- dialog boxes
 - creating **230**
 - predefined **225**
 - showing and hiding **231**
- dialog parameters
 - setting **396**
- display next window **406**
- display previous window **406**
- dockable containers
 - introducing **331**
 - orthogonal **335**
- docking bars **339**
 - customizing **341**
- docking panes
 - creating **333**
 - handles **334**
 - introducing **331**
- document
 - description **372**
 - general parameters **383**
 - setting selected parameters **385**
 - toolbar parameters **389**
- document parameters
 - setting **382**
- document type **379**
- Document/View architecture
 - classes **409**
 - description **371**
- doIt member function
 - IlvOptionMenu class **251**
 - IlvStringList class **262**
- drag member function
 - IlvScrollBar class **253**

- draw member function
 - IlvGadgetItem class **283**
- drawItem member function
 - IlvAbstractMatrix class **304**

E

- Edit Application icon **86**
- EditApplication command **40**
- editing
 - Application buffer windows **28**
 - applications **84**
 - menus **66**
 - objects **55**
 - pop-up menus **69**
- editing modes
 - Active **32, 59**
 - Attachments **33, 61**
 - Focus **33, 60**
 - for application buffer **33**
 - for gadgets buffer **32**
 - Generate **33**
 - initializing **151**
 - Label **32**
 - Label List **32**
 - Matrix **33, 75**
 - Menu **33, 71**
 - PolySelection **32**
 - Rotate **32**
 - Selection **32**
 - Spin Box **33**
- editing states **428**
- editing states example
 - chaining states **435**
 - changing the callback of the Show button **437**
 - changing the label **437**
 - creating a substate **438**
 - creating the first panel **429**
 - creating the second panel **430**
 - editing the Show state **433**
 - overview **428**
 - panel descriptions **432**
 - state file description **442**
- editItem member function
 - IlvMatrix class **314**

- editors **181**
 - list **172**
 - paired with accessors **181**
 - stand-alone **181**
 - tree **175**
- enableLargeList member function
 - IlvScrolledComboBox class **236**
- errors
 - See IlvStError class **143**
- exiting **406**
- ExpandCallbackType member function
 - IlvHierarchicalSheet class **319**
 - IlvTreeGadget class **276**
- expandItem member function
 - IlvTreeGadget class **273**
- extending IBM ILOG Studio **142**
 - example **156**
- extensions
 - commands **143**
 - panels **144**

F

- file
 - new **405**
 - operations menu **405**
- files
 - generated **89**
 - header **93, 101**
 - make **89**
 - source **93, 101**
- fitToSize member function
 - IlvMatrix class **307**
- flipLabels member function
 - IlvNotebook class **243**
- focus chain **60**
 - defining **205**
- focus management **204**
- Focus mode **33, 60**
- focusIn member function
 - IlvGadget class **205**
- focusOut member function
 - IlvGadget class **205**
- forcing
 - action state **427**

frame window **371**
frames **239**
 associating a mnemonic **239**
functions
 main **128**

G

gadget containers **213**
gadget holders **202, 203**
 IlvMouseEnter event **204**
 IlvMouseLeave event **204**
 limitations **204**
gadget items
 creating **285**
 drawing **283**
 finding **284**
 introducing **278**
 managing lists of **287**
 nonsensitive **282**
 palettes **283**
 represented by a bitmap **281**
 represented by a graphic object **281**
 setting a mnemonic **280**
 sorting **289**
gadgets
 arrow button **233**
 associating callbacks **210**
 associating mnemonics **212**
 attaching to guides **207**
 attachments **206**
 button **233**
 containers **203**
 creating **55**
 handling events **204**
 inside matrix **203**
 inside notebooks **203**
 inside tool bars **203**
 localizing **211**
 look and feel **218**
 Microsoft Windows look and feel **218**
 Motif look and feel **218**
 predefined callback types **210**
 sensitive **208**
 setting the weight **207**

 setting tooltips **212**
 thickness **209**
 transparent **209**
Gadgets buffer window **55**
generate all **404, 405**
Generate command **40**
generate data **404**
Generate mode **33**
GenerateAll command **41**
GenerateApplication command **41**
generated code
 extending **129**
generated files **89**
GenerateMakeFile command **41**
GeneratePanelClass command **42**
GeneratePanelSubClass command **42**
generating
 all **404**
 application **405**
 C++ code **121**
 callback method **124**
 custom **404, 405**
 data **404, 405**
 header files **121**
 parameters **401**
 specific files **405**
generation
 all **404**
 custom **404**
 data **404**
 operations menu **405**
 setting project parameters **401, 405**
generic document **380**
geometryChanged member function
 IlvAbstractBar class **299**
get class
 IlvPopupMenu class **297**
Get member function
 IlvDesktopManager class **352**
getBitmap member function
 IlvNotebookPage class **247**
getBitmapCount member function
 IlvGadgetItem class **281**
getCallbackItem member function
 IlvTreeGadget class **275**

getCardinal member function
 IlvPanedContainer class **325**
 getCheckColor member function
 IlvColoredToggle class **268**
 getColumnWidth member function
 IlvAbstractMatrix class **304**
 getCurrentBitmap member function
 IlvGadgetItem class **281**
 getCurrentLook member function
 IlvDisplay class **222**
 getCurrentState member function
 IlvViewFrame class **354**
 getDirection member function
 IlvArrowButton class **233**
 IlvPanedContainer class **324**
 getFileName member function
 IlvNotebookPage class **245**
 getFirstChild member function
 IlvTreeGadgetItem class **273**
 getFloatValue member function
 IlvNumberField class **249**
 IlvTextField class **265**
 getGuideCardinal member function
 IlvGraphicHolder class **207**
 getGuideLimit member function
 IlvGraphicHolder class **207**
 getGuidePosition member function
 IlvGraphicHolder class **207**
 getGuideSize member function
 IlvGraphicHolder class **207**
 getGuideWeight member function
 IlvGraphicHolder class **207**
 getHeight member function
 IlvGadgetItem class **282**
 getHighlightTextPalette member function
 IlvGadgetItemHolder class **283**
 getIncrement member function
 IlvSpinbox class **257**
 getIndex member function
 IlvPanedContainer class **325**
 getInsensitivePalette member function
 IlvGadgetItemHolder class **283**
 getIntValue member function
 IlvNumberField class **249**
 IlvTextField class **265**
 getItem member function
 IlvMatrix class **310**
 IlvMenuItem class **293**
 getItemByName member function
 IlvGadgetItemHolder class **284**
 getLabel member function
 IlvNotebookPage class **247**
 IlvOptionMenu class **251**
 getLabels member function
 IlvSpinbox class **257**
 getLabelsCount member function
 IlvSpinbox class **257**
 getLine member function
 IlvText class **263**
 getMinimizedFrames member function
 IlvDesktopManager class **355**
 getNextSibling member function
 IlvTreeGadgetItem class **273**
 getNormalTextPalette member function
 IlvGadgetItemHolder class **283**
 getNotebook member function
 IlvNotebook class **245**
 getOpaquePalette member function
 IlvGadgetItemHolder class **283**
 getOrientation member function
 IlvAbstractBar class **298**
 getPageArea member function
 IlvNotebook class **244**
 getPageBottomMargin member function
 IlvNotebook class **244**
 getPageLeftMargin member function
 IlvNotebook class **244**
 getPageRightMargin member function
 IlvNotebook class **244**
 getPages member function
 IlvNotebook class **245**
 getPagesCardinal member function
 IlvNotebook class **245**
 getPageTopMargin member function
 IlvNotebook class **244**
 getPane member function
 IlvPanedContainer class **325**
 getParent member function
 IlvTreeGadgetItem class **273**
 getPrevSibling member function

- IlvTreeGadgetItem class **273**
- getResources member function
 - IlvDisplay class **216**
- getRoot member function
 - IlvTreeGadget class **272**
- getRowHeight member function
 - IlvAbstractMatrix class **304**
- getSelectionMode member function
 - IlvTreeGadget class **276**
- getSelectionPalette member function
 - IlvGadgetItemHolder class **283**
- getSelectionTextPalette member function
 - IlvGadgetItemHolder class **283**
- getSpacing member function
 - IlvAbstractBar class **299**
- getState member function
 - IlvToggle class **268**
- getTabsPosition member function
 - IlvNotebook class **243**
- getText member function
 - IlvText class **263**
- getTreeItem member function
 - IlvHierarchicalSheet class **318**
- getValue member function **130**
 - IlvDateField class **238**
 - IlvSpinbox class **257**
- getView member function
 - IlvDesktopManager class **352**
 - IlvNotebookPage class **245**
- getViewPane member function
 - IlvPanedContainer class **326**
- getWidth member function
 - IlvGadgetItem class **282**
- getXMargin member function
 - IlvNotebook class **243**
- getYMargin member function
 - IlvNotebook class **243**
- grapher application **380**
- graphic objects
 - integrating **151**
- graphic panes
 - creating **323**
- GUI events
 - track and process **371, 422**

H

- handleEvent member function
 - IlvGadget class **204**
- handleMatrixEvent member function
 - IlvAbstractMatrix class **305**
 - IlvMatrix class **312**
- handling menu and toolbar items **411**
- header files **93, 121, 122, 125**
- headerFileExtension option **138**
- hide member function
 - IlvPane class **324**
- hierarchical sheets **317**
 - creating **317**
 - expanding or collapsing an item **318**
 - handling events **319**
 - navigating **318**
 - removing items **318**
- HighlightCBSymbol member function
 - IlvAbstractMenu class **291**
- HighlightedBitmapSymbol member function
 - IlvGadgetItem class **281**

I

- icons
 - Attachments **61**
 - Edit Application **86**
 - Inspect **58, 67**
 - Menu **71**
 - Panel Class Palette **96**
 - Test **59, 111**
- IlvAbstractBar class **298, 340**
 - defaultItemsSize member function **299**
 - geometryChanged member function **299**
 - getOrientation member function **298**
 - getSpacing member function **299**
 - setConstraintMode member function **299**
 - setDefaultItemSize member function **300**
 - setFlushingRight member function **300**
 - setOrientation member function **298**
 - setSpacing member function **299**
 - useConstraintMode member function **299**
- IlvAbstractBarPane class
 - checkLabelOrientation member function **342**

geometryChanged member function **341**
 orientationChanged member function **341**
IlvAbstractBarPane class **340**
IlvAbstractMatrix class **302**
 cellBBox member function **305**
 cellInfo member function **304**
 columnBBox member function **305**
 columns member function **303**
 columnSameWidth member function **303**
 drawItem member function **304**
 getColumnWidth member function **304**
 getRowHeight member function **304**
 handleMatrixEvent member function **305**
 invalidateColumn member function **305**
 invalidateRow member function **305**
 pointToPosition member function **305**
 rowBBox member function **305**
 rows member function **303**
 rowSameHeight member function **303**
 setNbFixedColumn member function **304**
 setNbFixedRow member function **304**
 subclassing **303**
IlvAbstractMenu class **291, 294**
 HighlightCBSymbol member function **291**
 isSelectable member function **292**
 select member function **292**
 selectNext member function **292**
 unSelect member function **292**
IlvApplication class **28, 90, 94, 126, 411**
 makePanels member function **126, 127**
IlvArrowButton class **233**
 getDirection member function **233**
 setDirection member function **233**
IlvBitmapMatrixItem class **307**
IlvButton class **233**
 activate member function **234**
 setHighlightedBitmap member function **234**
 setSelectedBitmap member function **234**
 showFrame member function **234**
IlvColoredToggle class
 getCheckColor member function **268**
 setCheckColor member function **268**
IlvColorSelector class **229**
IlvComboBox class **235**
 setSelected member function **236**
 whichSelected member function **236**
IlvContainer class **27**
IlvDateField class **237**
 getValue member function **238**
 setFormat member function **237**
 setValue member function **238**
IlvDesktopManager class
 Get member function **352**
 getMinimizedFrames member function **355**
 getView member function **352**
IlvDisplay class
 addChangeLookCallback member function **222**
 and gadgets look and feel **222**
 getCurrentLook member function **222**
 getResources member function **216**
 setCurrentLook member function **222**
IlvDockable class **331, 334, 338**
 acceptDocking member function **339**
 dock member function **338**
 isDocked member function **338**
 setDockable member function **339**
 setDockingDirection member function **339**
 undock member function **338**
IlvDockableContainer class **331**
 acceptDocking member function **339**
 addDockingPane member function **334, 338**
 createOrthogonalDockableContainer
 member function **336**
IlvDockableMainWindow class **345**
 addRelativeDockingPane member function **345**
IlvDoubleMatrixItem class **308**
IlvDvApplication class **409, 411**
 description **409**
 setAppOptionsFilename member function **410**
 setUserOptionsFilename member function **410**
IlvDvDocument class **409**
 description **409**
 initializeDocument member function **412**
IlvDvDocViewInterface class **417**
 description **409**
 initializeView member function **418**
IlvDvSerializable class **413**
IlvDvStream class **413**
IlvFilledDoubleMatrixItem class **308**
IlvFilledFloatMatrixItem class **308**

IlvFilledIntMatrixItem class 308
IlvFilledLabelMatrixItem class 307
IlvFloatMatrixItem class 308
IlvFontSelector class 229
IlvFrame class 239
IlvGadget class 197
 focusIn member function **205**
 focusOut member function **205**
 handleEvent member function **204**
 IlvMouseEnter event **204**
 IlvMouseLeave event **204**
 reDraw member function **235**
 setTransparent member function **209**
IlvGadgetContainer class 97, 99, 102, 202, 213
IlvGadgetItem class 271, 278
 BitmapSymbol member function **281**
 draw member function **283**
 getBitmapCount member function **281**
 getCurrentBitmap member function **281**
 getHeight member function **282**
 getWidth member function **282**
 HighlightedBitmapSymbol member function **281**
 InsensitiveBitmapSymbol member function **281**
 labelRect member function **282**
 pictureRect member function **282**
 SelectedBitmapSymbol member function **281**
 setBitmap member function **281**
 setGraphic member function **281, 300**
 setHighlightTextPalette member function **283**
 setLabel member function **280**
 setLabelAlignment member function **280**
 setLabelOrientation member function **280**
 setLabelPosition member function **281**
 setNormalTextPalette member function **283**
 setOpaquePalette member function **283**
 setSelectionTextPalette member function **283**
 setSensitive member function **282**
 setSpacing member function **281**
 showPicture member function **282**
IlvGadgetItemHolder class 283
 createItem member function **285**
 getHighlightTextPalette member function **283**
 getInsensitivePalette member function **283**
 getItemByName member function **284**
 getNormalTextPalette member function **283**
 getOpaquePalette member function **283**
 getSelectionPalette member function **283**
 getSelectionTextPalette member function **283**
 reDrawItems member function **285**
IlvGadgetItemMatrixItem class 308
IlvGadgetManager class 203, 213
IlvGadgetMatrixItem class 308
IlvGraphic class 213
 applyResources member function **216**
 setFirstFocusGraphic member function **205**
 setLastFocusGraphic member function **205**
 setNextFocusGraphic member function **205**
 setPreviousFocusGraphic member function **205**
IlvGraphicCallback function 100
IlvGraphicHolder class
 addGuide member function **207**
 attach member function **207**
 getGuideCardinal member function **207**
 getGuideLimit member function **207**
 getGuidePosition member function **207**
 getGuideSize member function **207**
 getGuideWeight member function **207**
 removeGuide member function **207**
IlvGraphicMatrixItem class 308
IlvGraphicPane class 323
IlvGraphicSet class 270
IlvHierarchicalSheet class 302, 317
 ExpandCallbackType member function **319**
 getTreeItem member function **318**
 ShrinkCallbackType member function **319**
IlvIErrorDialog class 226
IlvIFileSelector class 227
IlvIInformationDialog class 227
IlvIMessageDialog class 225
IlvIntMatrixItem class 308
IlvIPromptString class 228
IlvIQuestionDialog class 226
IlvIWarner class 227
IlvLabelMatrixItem class 307
IlvListGadgetItemHolder class 287, 291
 compareItems member function **289**
 insertGraphic member function **300**
 sort member function **289**
IlvManager class 27, 145
IlvManagerViewInteractor class 146

IlvMatrix class 302, 305
 activateMatrixItem member function **314**
 adjustLast member function **307**
 allowEdit member function **313**
 autoFitToSize member function **307**
 editItem member function **314**
 fitToSize member function **307**
 getFirstSelected member function **311**
 getItem member function **310**
 handleMatrixEvent member function **312**
 insertColumn member function **306**
 insertRow member function **306**
 pointToItem member function **315**
 pointToPosition member function **315**
 reinitialize member function **306**
 remove member function **310**
 removeColumn member function **306**
 removeRow member function **306**
 resizeColumn member function **306**
 resizeRow member function **306**
 sameHeight member function **307**
 sameWidth member function **307**
 set member function **310**
 setBrowseMode member function **313**
 setColumnSelected member function **311**
 setExclusive member function **312**
 setItemAlignment member function **311**
 setItemCallback member function **314**
 setItemReadOnly member function **312**
 setItemRelief member function **311**
 setItemSelected member function **311**
 setItemSensitive member function **312**
 setXGrid member function **306**
 setYGrid member function **306**
 showLabel member function **315**
 showPicture member function **315**
 validate member function **314**
IlvMenuBar class 300
IlvMenuItem class 292
 getItem member function **293**
 setMenu member function **293**
 setTooltip member function **301**
IlvMessageLabel class
 setAlignment member function **241**
 setBitmap member function **240**
 setInsensitiveBitmap member function **240**
 setLabelPosition member function **241**
 setSpacing member function **241**
 setTransparent member function **241**
IlvMessageLabel code 239
IlvNotebook class 242
 addPage member function **244**
 areLabelsVertical member function **243**
 changeSelection member function **247**
 flipLabels member function **243**
 getNotebook member function **245**
 getPageArea member function **244**
 getPageBottomMargin member function **244**
 getPageLeftMargin member function **244**
 getPageRightMargin member function **244**
 getPages member function **245**
 getPagesCardinal member function **245**
 getPageTopMargin member function **244**
 getTabsPosition member function **243**
 getXMargin member function **243**
 getYMargin member function **243**
 pageDeselected member function **247**
 PageResizeCallbackType member function **247**
 pageSelected member function **247**
 removePage member function **245**
 setLabelsVertical member function **243**
 setPageBottomMargin member function **244**
 setPageRightMargin member function **244**
 setPageTopMargin member function **244**
 setTabsPosition member function **243**
 setXMargin member function **243**
 setYMargin member function **243**
IlvNotebookPage class 242, 244
 getBitmap member function **247**
 getFileName member function **245**
 getLabel member function **247**
 getView member function **245**
 mustFlipLabels member function **243**
 setBackground member function **246**
 setBitmap member function **247**
 setFileName member function **245**
 setLabel member function **247**
IlvNumberField class 247
 getFloatValue member function **249**
 getIntValue member function **249**

- setDecimalPointChar member function **250**
- setFormat member function **248**
- setMaxFloat member function **249**
- setMaxInt member function **249**
- setMinFloat member function **249**
- setMinInt member function **249**
- setThousandSeparator member function **249**
- setValue member function **249**
- validate member function **250**
- IlvOptionMenu class **250**
 - doIt member function **251**
 - getLabel member function **251**
 - setSelected member function **251**
 - whichSelected member function **251**
- IlvPane class **320**
 - hide member function **324**
 - predefined subclasses **323**
 - show member function **324**
- IlvPanedContainer class **203, 320, 324**
 - addPane member function **325**
 - createSliderPane member function **328**
 - getCardinal member function **325**
 - getDirection member function **324**
 - getIndex member function **325**
 - getPane member function **325**
 - getViewPane member function **326**
 - manageSliders member function **328**
 - removePane member function **325**
 - setDirection member function **324**
 - setMinimumSize member function **327**
 - setResizeMode member function **326**
 - updatePanels member function **324, 325**
- IlvPasswordField class **251**
 - setMaskChar member function **251**
- IlvPopupMenu class **294**
 - get class **297**
 - isChecked class **297**
 - OpenMenuCallbackSymbol class **296**
 - setChecked class **297**
 - setLabelOffset class **295**
 - setTearOff class **296**
- IlvScrollBar class **252**
 - drag member function **253**
 - setIncrement member function **252**
 - setPageIncrement member function **252**

- setValues member function **252**
- valueChanged member function **253**
- IlvScrollbar class **252**
 - setOrientation member function **252**
- IlvScrolledComboBox class **235**
 - enableLargeList member function **236**
 - setVisibleItems member function **236**
- IlvSelector class **270**
 - whichGraphicSelected member function **270**
 - whichSelected member function **270**
- IlvSheet class **302, 316**
- IlvSimpleGraphic class **214**
- IlvSlider class
 - setOrientation member function **254**
 - setPageIncrement member function **254**
 - setValues member function **254**
 - valueChanged member function **255**
- IlvSliderPane class **327**
- IlvSpinBox class **255**
 - addField member function **256**
 - addLabel member function **257**
 - addObject member function **257**
 - DecrementCallbackType member function **258**
 - getIncrement member function **257**
 - getLabels member function **257**
 - getLabelsCount member function **257**
 - getValue member function **257**
 - IncrementCallbackType member function **258**
 - numeric fields **257**
 - removeLabel member function **257**
 - removeObject member function **257**
 - setIncrement member function **257**
 - setValue member function **257**
 - text fields **257**
- IlvStBuffer class **145**
- IlvStClickAddObject class
 - redefining **151**
- IlvStCommand class **143**
- IlvStContainerInfo class
 - description of **146**
- IlvStDialog class **144**
- IlvStError class **143**
- IlvStGadgetBuffer class **145**
- IlvStIAccessor class **169**
 - apply member function **169**

- initialize member function **169**
- IlvStICallbackPrecondition class **179**
- IlvStICombinedAccessor class **172**
- IlvStIEditor class **181**
- IlvStIError class **180**
- IlvStIMainEditor class **165**
- IlvStInspector class **164**
- IlvStInspectorPanelBuilder class **150**
- IlvStIPrecondition class **179**
- IlvStIPreconditionValue class **179**
- IlvStIProperty class **170**
- IlvStIPropertyAccessor class **169**
- IlvStIPropertyListAccessor class **172**
- IlvStIPropertyListEditor class **172**
- IlvStIPropertyTreeAccessor class **175**
- IlvStIPropertyTreeEditor class **175**
- IlvStIRangeValidator class **180**
- IlvStIValidator class **179, 180**
- IlvStIValueProperty class **170**
- IlvStMode class **146**
- IlvStPanelHandler class **144**
- IlvStringList class **258**
 - autoLabelAlignment member function **260**
 - doIt member function **262**
 - select member function **262**
 - setDefaultItemHeight member function **259**
 - setExclusive member function **262**
 - setLabelOffset member function **260**
 - setLabelPosition member function **259**
 - setSelectCallback member function **262**
 - setSelected member function **262**
 - setSelectionLimit member function **262**
 - setSelectionMode member function **262**
 - showLabel member function **259**
 - showPicture member function **259**
 - unSelect member function **262**
 - useFullSelection member function **260**
 - useToolTips member function **260**
- IlvStSubscription class **144**
- IlvText class **262**
 - addLine member function **263**
 - check member function **263**
 - getLine member function **263**
 - getText member function **263**
 - removeLine member function **263**

- setLine member function **263**
- setText member function **263**
- IlvTextField class **264**
 - check member function **266**
 - getFloatValue member function **265**
 - getIntValue member function **265**
 - labelChanged member function **266**
 - setAlignment member function **265**
 - setChangeCallback member function **266**
 - setEditable member function **235**
 - setMaxChar member function **265**
 - setValue member function **265**
 - validate member function **266**
- IlvToggle class **267**
 - getState member function **268**
 - setBitmap member function **269**
 - setCheckSize member function **269**
 - setPosition member function **269**
 - setRadio member function **268**
 - setState member function **268**
- IlvToolBar class **300**
 - useToolTips member function **301**
- IlvTreeGadget class **271**
 - ActivateCallbackType member function **276**
 - addItem member function **272, 273**
 - detachItem member function **273**
 - ExpandCallbackType member function **276**
 - expandItem member function **273**
 - getCallbackItem member function **275**
 - getRoot member function **272**
 - getSelectionMode member function **276**
 - removeAllItems member function **273**
 - SelectCallbackType member function **276**
 - setSelectionMode member function **276**
 - showLines member function **274**
 - ShrinkCallbackType member function **276**
 - shrinkItem member function **273**
- IlvTreeGadgetItem class **271**
 - getFirstChild member function **273**
 - getNextSibling member function **273**
 - getParent member function **273**
 - getPrevSibling member function **273**
 - insertChild member function **273**
 - setUnknownChildCount member function **273**
- IlvViewFrame class **351**

- getCurrentState member function **354**
- maximizeFrame member function **356**
- minimizeFrame member function **355**
- restoreFrame member function **355**
- setIcon member function **353**
- setTitle member function **353**
- IlvViewPane class **323**
- implementation
 - new document **412**
 - serialization **412**
- IncrementCallbackType member function
 - IlvSpinBox class **258**
- inheritance tree **412**
 - document view class **418**
- initialize member function **125**
- initializeDocument member function
 - IlvDvDocument class **412**
- initializeView member function
 - IlvDvDocViewInterface class **418**
- initializing an inspector panel **165**
- InsensitiveBitmapSymbol member function
 - IlvGadgetItem class **281**
- insertChild member function
 - IlvTreeGadgetItem class **273**
- insertColumn member function
 - IlvMatrix class **306**
- insertGraphic member function
 - IlvListGadgetItemHolder class **300**
- insertRow member function
 - IlvMatrix class **306**
- Inspect icon **58, 67**
- inspecting
 - matrix items **76**
 - objects **58**
- inspector panels **58**
 - accessors **165**
 - components **165**
 - defining **154**
 - editors **165**
 - initialization steps **165**
- inspectors
 - Application **49, 89**
 - definition **164**
 - initializing **150**
 - Panel Class **98**

- State **432**
- interactions **418**
- interface declaration
 - sample **423**
- interface mechanism **422**
- introspection **422**
- invalidateColumn member function
 - IlvAbstractMatrix class **305**
- invalidateRow member function
 - IlvAbstractMatrix class **305**
- isChecked class
 - IlvPopupMenu class **297**
- isFirstSelected member function
 - IlvMatrix class **311**
- isSelectable member function
 - IlvAbstractMenu class **292**
- istream **413**

J

- JvScriptApplication option **138**

K

- keyboard focus **204**
- keyboard focus chain
 - description of **60**
- KillTestPanels command **42**

L

- Label List mode **32**
- Label mode **32**
- labelChanged member function
 - IlvTextField class **266**
- labelRect member function
 - IlvGadgetItem class **282**
- labels **239**
- list accessors **172**
- list editors **172**
- loading
 - ODV file **405**
- loading data **413**
- look and feel **217**
 - changing **217**

Motif **217**
Windows **217**

M

macros

action **427**
naming conventions **423**
RefreshAction **427**

main file

description **411**
sample code **411**

main function **128**

Main window title **381**

make file **89, 132**

MakeDefaultApplication command **43**

makeFileExtension option **138**

makePanels member function **126**

manager application **380**

manageSliders member function

 IlvPanedContainer class **328**

managing options data **411**

manual

naming conventions **21**
notation **21**

matrices

adding rows and columns **306**
drawing items on multiple cells **304**
fit to size mode **307**
fixed rows and columns **304**
handling events **305, 312**
number of columns **306**
number of rows **306**
resizing rows and columns **306**
reverting to initial settings **307**
selection modes **312**
tooltips **316**
using gadget items **315**
using gadgets **315**

matrix items

adding **310**
aligning **310**
bitmap images **307**
callbacks **314**
double-precision floating point values **308**

editing **313**

filled double-precision floating point values **308**

filled floating point values **308**

filled integers **308**

filled labels **307**

floating point values **308**

gadget items **308**

gadgets **308, 315**

graphic objects **308**

in relief **311**

integers **308**

labels **307**

predefined classes **307**

redrawing **310**

removing **310**

selecting **311**

sensitive **312**

Matrix mode **33, 75**

 extracting matrix items **76**

 inspecting matrix items **76**

 setting up matrix items **75**

maximizeFrame member function

 IlvViewFrame class **356**

menu bars **291**

 constraining the geometry **299**

 creating **66**

 default item size **299**

 flushing items **300**

 notifying geometry change **299**

 setting the orientation **298**

Menu icon **71**

menu items **292**

 accelerators **294**

 adding **381**

 adding a submenu **293**

 associating mnemonics **294**

 attaching a callback **293**

 callbacks **296**

 check marks **296**

 checked **296**

 creating **292**

 manipulating **291**

 used as separator **292**

Menu mode **33**

menu separator **292**

- menus
 - attaching pop-up menus **71**
 - callbacks **291**
 - creating popup **395**
 - creating popup submenus **396**
 - editing **66**
 - handling events **292**
 - introducing **290**
- message labels **239**
 - bitmaps **240**
 - laying out **241**
 - localizing **242**
 - mnemonics **242**
 - opaque **241**
 - transparent **241**
- messages **144**
- minimizeFrame member function
 - IlvViewFrame class **355**
- modules
 - insert and remove **405**
- mustFlipLabels member function
 - IlvNotebookPage class **243**

N

- naming conventions **21**
 - examples **424**
 - fields **424**
 - macros **423**
 - methods **423**
- new application **375**
- new project **405**
- NewApplication command **28, 43**
- NewGadgetBuffer command **43**
- NewGraphicBuffer command **28**
- NewPanelClass command **44**
- noPanelContents option **139**
- notation **21**
- notebook pages **242**
- notebooks **242**
 - callbacks **247**
 - customizing pages **245**
 - handling events **247**
 - page color **246**
 - tab content **247**

- tab margins **243**
- tab orientation **243**
- tab position **243**
- tabs content **247**

- number fields **247**
 - callbacks **250**
 - decimal point **250**
 - editing modes **248**
 - retrieving values **249**
 - setting values **249**
 - thousand separator **249**

O

- object files
 - linking **132**
- object resources
 - adding **216**
 - predefined **213**
 - priority **215**
 - setting **214**
- objects
 - activating **59**
 - attaching **61**
 - creating **55**
 - editing **55**
 - inspecting **58**
 - setting the focus **60**
 - using creation mode **56**
- objects, creating **55**
- ODV file
 - closing **405**
 - loading **405**
 - saving **405**
- OpenApplication command **44**
- opening
 - application **88, 375**
 - ODV file **405**
- OpenMenuCallbackSymbol class
 - IlvPopupMenu class **296**
- option files
 - application **410**
 - Application Framework **410**
 - description **410**
 - user **410**

- option menus **250**
 - callbacks **251**
 - localizing **251**
 - retrieving items **251**
 - selected item **251**
 - setting items **251**
- options file **380**
- ostream **413**

P

- pageDeselected member function
 - IlvNotebook class **247**
- PageResizeCallbackType member function
 - IlvNotebook class **247**
- pageSelected member function
 - IlvNotebook class **247**
- palette **376**
- Palettes panel
 - customizing **153**
- paned containers **204, 320**
 - creating **324**
 - direction **324**
 - encapsulating in a view pane **325**
 - modifying the layout **325**
- Panel Class inspector
 - description of **98**
- Panel Class palette **95**
 - accessing **96**
 - commands **97**
 - description of **29, 87, 96**
- Panel Class Palette icon **96**
- panel classes **145**
 - adding **97**
 - adding an instance **102**
 - callback declarations **100**
 - callback definitions **101**
 - creating **97, 116**
 - creating instances **102**
 - general properties **99**
 - header files **101**
 - managing panel instances **103**
 - removing **98**
 - setting options **99**
 - source files **101**

- Panel inspector **104**
- panel instances
 - creating **116, 117**
- panelBaseClass option **139**
- panels
 - accessors **127**
 - adding **144**
 - Application inspector **49**
 - Attachments **63**
 - creating **55, 114**
 - initializing **150**
 - inspecting **104**
 - menu bar inspector panel **67**
 - State Inspector **432**
 - State Tree **432**
 - Test **59**
 - testing **59**
- panes
 - creating **323**
 - docking **338**
 - elastic **326**
 - fixed **326**
 - hiding **324**
 - introducing **320**
 - minimum size **326**
 - resizable **326**
 - resizing **326**
 - resizing mode **326**
 - retrieving **325**
 - showing **324**
 - sliders **327**
 - undocking **338**
- parameters
 - action **389**
 - application **380**
 - data **400**
 - dialog **396**
 - document **382**
 - general document **383**
 - generating **401**
 - popup **393**
 - selected document **385**
 - toolbar for a document **389**
 - window **386**
- passwords **251**

- pictureRect member function
 - IlvGadgetItem class **282**
- pointToItem member function
 - IlvMatrix class **315**
- pointToPosition member function
 - IlvAbstractMatrix class **305**
 - IlvMatrix class **315**
- PolySelection mode **32**
- popup
 - creating menu **395**
 - creating submenu **396**
 - setting parameters **393**
- popup menu items
 - adding **396**
- pop-up menus **291, 294**
 - aligning items **295**
 - attaching **71**
 - contextual **297**
 - creating **69**
 - editing **69**
 - tear-off **296**
- processing
 - GUI events **422**
- profile options file
 - Unix **410**
 - Windows **410**
- project application **380**
- property accessors **169**

Q

- quitting
 - Application Framework Editor **406**
- quotation marks **413**

R

- radio buttons **267, 268**
 - grouping **270**
- reDraw member function
 - IlvGadget class **235**
- reDrawItems member function
 - IlvGadgetItemHolder class **285**
- reinitialize member function
 - IlvMatrix class **306**

- remove member function
 - IlvMatrix class **310**
- RemoveAllAttachments command **44**
- removeAllItems member function
 - IlvTreeGadget class **273**
- RemoveAttachments command **44**
- removeColumn member function
 - IlvMatrix class **306**
- removeGuide member function
 - IlvGraphicHolder class **207**
- removeLabel member function
 - IlvSpinbox class **257**
- removeLine member function
 - IlvText class **263**
- removeObject member function
 - IlvSpinBox class **257**
- removePage member function
 - IlvNotebook class **245**
- removePane member function
 - IlvPanedContainer class **325**
- RemovePanel command **45**
- RemovePanelClass command **45**
- removeRow member function
 - IlvMatrix class **306**
- resizeColumn member function
 - IlvMatrix class **306**
- resizeRow member function
 - IlvMatrix class **306**
- resources
 - gadgets **213**
 - objects **213**
 - predefined **213**
- restoreFrame member function
 - IlvViewFrame class **355**
- Rotate mode **32**
- rowBBox member function
 - IlvAbstractMatrix class **305**
- rows member function
 - IlvAbstractMatrix class **303**
- rowSameHeight member function
 - IlvAbstractMatrix class **303**

S

- sameHeight member function

- IlvMatrix class **307**
- sameWidth member function
 - IlvMatrix class **307**
- SaveApplication command **45**
- SaveApplicationAs command **45**
- saving
 - applications **88**
 - ODV file **405**
- saving as
 - ODV file **405**
- saving data **413**
- script project **405**
- script project file **405**
- scrollbars **252**
 - callbacks **253**
 - increment **252**
 - page increment **252**
 - values **252**
- scrolled combo boxes
 - large lists **236**
 - number of visible items **236**
- select member function
 - IlvAbstractMenu class **292**
 - IlvStringList class **262**
- SelectAttachmentsMode command **46**
- SelectCallbackType member function
 - IlvTreeGadget class **276**
- SelectedBitmapSymbol member function
 - IlvGadgetItem class **281**
- SelectFocusMode command **46**
- Selection mode **32**
- SelectMatrixMode command **46**
- SelectMenuMode command **47**
- selectNext member function
 - IlvAbstractMenu class **292**
- set member function
 - IlvMatrix class **310**
- setAlignment member function
 - IlvMessageLabel class **241**
 - IlvTextField class **265**
- setAppOptionsFilename member function
 - IlvDvApplication class **410**
- setBackground member function
 - IlvNotebookPage class **246**
- setBitmap member function

- IlvGadgetItem class **281**
- IlvMessageLabel class **240**
- IlvNotebookPage class **247**
- IlvToggle class **269**
- setBrowseMode member function
 - IlvMatrix class **313**
- setChangeCallback member function
 - IlvTextField class **266**
- setCheckColor member function
 - IlvColoredToggle class **268**
- setChecked class
 - IlvPopupMenu class **297**
- setCheckSize member function
 - IlvToggle class **269**
- setColumnSelected member function
 - IlvMatrix class **311**
- setConstraintMode member function
 - IlvAbstractBar class **299**
- setCurrentLook member function
 - IlvDisplay class **222**
- setDecimalPointChar member function
 - IlvNumberField class **250**
- setDefaultItemHeight member function
 - IlvStringList class **259**
- setDefaultItemSize member function
 - IlvAbstractBar class **299**
- setDirection member function
 - IlvArrowButton class **233**
 - IlvPanedContainer class **324**
- setEditable member function
 - IlvTextField class **235**
- setExclusive member function
 - IlvMatrix class **312**
 - IlvStringList class **262**
- setFileName member function
 - IlvNotebookPage class **245**
- setFirstFocusGraphic member function
 - IlvGraphic class **205**
- setFlushingRight member function
 - IlvAbstractBar class **300**
- setFormat member function
 - IlvDateField class **237**
 - IlvNumberField class **248**
- setGraphic member function
 - IlvGadgetItem class **281, 300**

setHighlightedBitmap member function
 IlvButton class **234**
 setHighlightTextPalette member function
 IlvGadgetItem class **283**
 setIcon member function
 IlvViewFrame class **353**
 setIncrement member function
 IlvScrollBar class **252**
 IlvSpinbox class **257**
 setInsensitiveBitmap member function
 IlvMessageLabel class **240**
 setItemAlignment member function
 IlvMatrix class **311**
 setItemCallback member function
 IlvMatrix class **314**
 setItemReadOnly member function
 IlvMatrix class **312**
 setItemRelief member function
 IlvMatrix class **311**
 setItemSelected member function
 IlvMatrix class **311**
 setItemSensitive member function
 IlvMatrix class **312**
 setLabel member function
 IlvGadgetItem class **280**
 IlvNotebookPage class **247**
 setLabelAlignment member function
 IlvGadgetItem class **280**
 setLabelOffset class
 IlvPopupMenu class **295**
 setLabelOffset member function
 IlvStringList class **260**
 setLabelOrientation member function
 IlvGadgetItem class **280**
 setLabelPosition member function
 IlvGadgetItem class **281**
 IlvMessageLabel class **241**
 IlvStringList class **259**
 setLabelsVertical member function
 IlvNotebook class **243**
 setLastFocusGraphic member function
 IlvGraphic class **205**
 setLine member function
 IlvText class **263**
 setMaskChar member function
 IlvPasswordTextField class **251**
 setMaxChar member function
 IlvTextField class **265**
 setMaxFloat member function
 IlvNumberField class **249**
 setMaxInt member function
 IlvNumberField class **249**
 setMenu member function
 IlvMenuItem class **293**
 setMinFloat member function
 IlvNumberField class **249**
 setMinimumSize member function
 IlvPanedContainer class **327**
 setMinInt member function
 IlvNumberField class **249**
 setNbFixedColumn member function
 IlvAbstractMatrix class **304**
 setNbFixedRow member function
 IlvAbstractMatrix class **304**
 setNextFocusGraphic member function
 IlvGraphic class **205**
 setNormalTextPalette member function
 IlvGadgetItem class **283**
 setOpaquePalette member function
 IlvGadgetItem class **283**
 setOrientation member function
 IlvAbstractBar class **298**
 IlvScrollbar class **252**
 IlvSlider class **254**
 setPageBottomMargin member function
 IlvNotebook class **244**
 setPageIncrement member function
 IlvScrollBar class **252**
 IlvSlider class **254**
 setPageRightMargin member function
 IlvNotebook class **244**
 setPosition member function
 IlvToggle class **269**
 setPreviousFocusGraphic member function
 IlvGraphic class **205**
 setRadio member function
 IlvToggle class **268**
 setResizeMode member function
 IlvPanedContainer class **326**
 setSelectCallback member function

- IlvStringList class **262**
- setSelected member function
 - IlvComboBox class **236**
 - IlvOptionMenu class **251**
 - IlvStringList class **262**
- setSelectedBitmap member function
 - IlvButton class **234**
- setSelectionLimit member function
 - IlvStringList class **262**
- setSelectionMode member function
 - IlvStringList class **262**
 - IlvTreeGadget class **276**
- setSelectionTextPalette member function
 - IlvGadgetItem class **283**
- setSensitive member function
 - IlvGadgetItem class **282**
- setSpacing member function
 - IlvAbstractBar class **299**
 - IlvGadgetItem class **281**
 - IlvMessageLabel class **241**
- setState member function
 - IlvToggle class **268**
- setTabsPosition member function
 - IlvNotebook class **243**
- setTearOff class
 - IlvPopupMenu class **296**
- setText member function
 - IlvText class **263**
- setThousandSeparator member function
 - IlvNumberField class **249**
- setting
 - action parameters **389**
 - application parameters **380**
 - data parameters **400**
 - dialog parameters **396**
 - document parameters **382**
 - document toolbar parameters **389**
 - general document parameters **383**
 - popup parameters **393**
 - project generation parameters **401, 405**
 - selected document parameters **385**
 - window parameters **386**
- setTitle member function
 - IlvViewFrame class **353**
- setTooltip member function

- IlvMenuItem class **301**
- setTransparent member function
 - IlvGadget class **209**
 - IlvMessageLabel class **241**
- setUnknownChildCount member function
 - IlvTreeGadgetItem class **273**
- setUserOptionsFilename member function
 - IlvDvApplication class **410**
- setValue member function
 - IlvDateField class **238**
 - IlvNumberField class **249**
 - IlvSpinbox class **257**
 - IlvTextField class **265**
- setValues member function
 - IlvScrollBar class **252**
 - IlvSlider class **254**
- setVisibleItems member function
 - IlvScrolledComboBox class **236**
- setXGrid member function
 - IlvMatrix class **306**
- setXMargin member function
 - IlvNotebook class **243**
- setYGrid member function
 - IlvMatrix class **306**
- setYMargin member function
 - IlvNotebook class **243**
- sheets **316**
- show member function
 - IlvPane class **324**
- ShowAllTestPanels command **47**
- ShowApplicationInspector command **47, 50, 52, 53**
- ShowClassPalette command **48**
- showFrame member function
 - IlvButton class **234**
- showLabel member function
 - IlvMatrix class **315**
 - IlvStringList class **259**
- showLines member function
 - IlvTreeGadget class **274**
- ShowPanelClassInspector command **48**
- showPicture member function
 - IlvGadgetItem class **282**
 - IlvMatrix class **315**
 - IlvStringList class **259**

ShrinkCallbackType member function

 IlvHierarchicalSheet class **319**

 IlvTreeGadget class **276**

shrinkItem member function

 IlvTreeGadget class **273**

slider panes **327**

sliders

 callbacks **255**

 increment **254**

 page increment **254**

 setting the orientation **252, 254**

 values **254**

sort member function

 IlvGadgetListItemHolder class **289**

source files **93, 124, 127**

sourceFileExtension option **139**

Spin box mode **33**

spin boxes **255**

 adding fields **256**

 adding graphic objects **257**

 callbacks **258**

 removing fields **257**

 with numeric fields **257**

 with text fields **257**

starting

 Application Framework Editor **374**

 new window **406**

state file **442**

State inspector

 description of **432**

State Tree panel

 description of **432**

states

 editing **428**

string lists **258**

 displaying items **259**

 drag-and-drop **262**

 editing items **262**

 full selection mode **260**

 gadget items **258**

 handling events **261**

 label alignment **259**

 localizing **261**

 multiple selection **261**

 partial selection mode **260**

 selection modes **261**

 single selection **261**

 tooltips **260**

strings

 blank spaces **413**

subpanels

 editing **108**

 inspecting **110**

system option **139**

T

Test icon **59, 111**

Test panel **59**

TestApplication command **48**

TestDocument command **48**

testing

 applications **111, 129**

 attachments **66**

testing panels **59**

TestPanel command **49**

text **262**

 handling events **263**

 retrieving **263**

 setting **263**

 special keys **263**

text application **380**

text fields **264**

 aligning **265**

 callbacks **266**

 handling events **266**

 keyboard shortcuts **267**

 localizing **265**

 number of characters **265**

tiling

 windows horizontally **406**

 windows vertically **406**

toggle buttons **267**

 bitmaps **269**

 callbacks **270**

 grouping **270**

 handling events **270**

 in color **267**

 localizing **269**

 mnemonics **269**

- position **269**
- state of **268**
- styles **268**
- text alignment **269**
- toolbar
 - Application Framework Editor **377**
- toolbar items
 - adding **382**
- toolBarItem option **140**
- toolbars **291**
 - constraining the geometry **299**
 - default item size **299**
 - docking **300**
 - flushing items **300**
 - managing gadgets **300**
 - notifying geometry change **299**
 - setting the orientation **298**
 - using tooltips **301**
- tools
 - menu **405**
- tooltips
 - attaching to gadgets **213**
 - creating **213**
 - enabling/disabling **213**
 - matrix **316**
 - string lists **260**
- tracking
 - GUI events **422**
- tree
 - accessors **175**
 - editors **175**
- tree gadgets **271**
 - callbacks **275**
 - collapsing items **273**
 - create items **272**
 - customizing **274**
 - drag-and-drop **276**
 - editing items **276**
 - expanding items **273**
 - moving items **273**
 - removing items **273**
 - scrollbars **272**
 - selection modes **275**

U

- unSelect member function
 - IlvAbstractMenu class **292**
 - IlvStringList class **262**
- updatePanels member function
 - IlvPanedContainer class **324, 325**
- useConstraintMode member function
 - IlvAbstractBar class **299**
- useFullSelection member function
 - IlvStringList class **260**
- user classes
 - setting up **130**
- userSubClassPrefix option **140**
- userSubClassSuffix option **140**
- useToolTips member function
 - IlvStringList class **260**
 - IlvToolBar class **301**
- Using Matrices on page 78 **33, 47**

V

- validate member function
 - IlvMatrix class **314**
 - IlvNumberField class **250**
 - IlvTextField class **266**
- valueChanged member function
 - IlvScrollBar class **253**
 - IlvSlider class **255**
- view frames **353**
 - changing the menu **353**
 - states **354**
- view panes
 - creating **323**
- views
 - client **350**

W

- whichGraphicSelected member function
 - IlvSelector class **270**
- whichSelected member function
 - IlvComboBox class **236**
 - IlvOptionMenu class **251**
 - IlvSelector class **270**

window
 display next **406**
 display previous **406**
 menu **405**
window parameters
 setting **386**
windows
 2D Graphics **27**
 Application **28, 84**
 cascade **406**
 Gadgets **55**
 starting new **406**
 tiling horizontally **406**
 tiling vertically **406**
workspace **377**

