IBM ILOG Views
2D Graphics 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** 12

What You Need to Know 12

Manual Organization 12

Notation 13

Typographic Conventions 13

Naming Conventions 13

Part I **Managers** 14

Chapter 1 **Basic Manager Features** 16

Introducing Managers 16

Layers 17

Views 18

View Transformer 18

Event Handling 18

Main Features of IlvManager 19

Manager Views 20

View Transformations 22

Double-buffering 22

Manager Layers **23**

Layer Index 24

Layer Selectability 25

Layer Visibility 25

Layer Rendering. 26

Managing Objects **27**

Modifying the Geometry of Graphic Objects 27

Selecting Objects 28

Selection Procedures 29

Managing Selected Objects 29

Managing Object Properties. 30

Arranging Objects 30

Drawing and Redrawing **32**

Optimizing Drawing Tasks **33**

Saving and Reading. **34**

Chapter 2 Manager Event Handling **36**

The Event Handling Mechanism. **36**

Event Hooks **37**

View Interactors **37**

Predefined View Interactors 38

Example: Implementing the IlvDragRectangleInteractor Class 39

Example of an Extension: IlvMoveInteractor 45

Object Interactors **52**

Accelerators **52**

Example: Changing the Key Assigned to an Accelerator. 53

Predefined Manager Accelerators 53

Chapter 3 Advanced Manager Features. **56**

Observers **56**

General Notifications 57

Manager View Notifications 57

Manager Layer Notifications 58
Manager Contents Notifications 59
Graphic Object Geometry Notifications 59
Example 59
View Hooks 60
Manager View Hooks 61
Example: Monitoring the Number of Objects in a Manager 62
Example: Maintaining a Scale Displayed With No Transformation 62
Manager Grid 64
Example: Using a Grid 65
Undoing and Redoing Actions 66
Command Class. 66
Managing Undo 66
Example: Using the IlvManagerCommand Class to Undo/Redo 67
Managing Modifications 68

Part II Grapher 70

Chapter 4 Introducing the Grapher Extension of IBM ILOG Views Studio 72

The Main Window 72
Buffer Windows 73
The Menu Bar 74
The Action Toolbar 75
The Editing Modes Toolbar 75
The Palettes Panel 75
The Grapher Palettes 76
Grapher Extension Commands 79
MakeNode 79
NewGrapherBuffer 79
SelectArcLinkImageMode 79
SelectDoubleLinkImageMode 80
SelectDoubleSplineLinkImageMode 80

SelectLinkImageMode 80
SelectOneLinkImageMode 81
SelectOneSplineLinkImageMode 81
SelectOrientedArcLinkImageMode 81
SelectOrientedDoubleLinkImageMode 81
SelectOrientedDoubleSplineLinkImageMode 82
SelectOrientedLinkImageMode 82
SelectOrientedOneLinkImageMode 82
SelectOrientedOneSplineLinkImageMode 83
SelectOrientedPolylineLinkImageMode 83
SelectPinEditorMode 83
SelectPolylineLinkImageMode 83

Chapter 5 Features of the Grapher Package 86

Graph Management 86
Description of the IlvGrapher Class 87
Loading and Saving Graph Descriptions 88
Grapher Links 89
Base Class for Links 89
Predefined Grapher Links 91
Creating a Custom Grapher link 96
Connection Pins 98
Grapher Interactors 101
Selection Interactor 101
Creating Nodes 102
Creating Links 102
Editing Connection Pins 104
Editing Links 104

Part III Prototypes 106

Chapter 6 Introducing the Prototypes Package 108

An Overview of the Prototypes Package **108**

Business Graphic Objects 109

Creating BGOs Using the Prototypes Extension of IBM ILOG Views Studio 110

Using Prototypes in Applications 110

When Should You Use Prototypes? 111

The Prototype Design Pattern 111

Specifying Graphic and Interactive Behavior Using Accessors 112

Chapter 7 Using IBM ILOG Views Studio to Create BGOs **114**

Creating and Using Prototypes **115**

Creating a Prototype Library 115

Creating a Prototype 115

Defining the Attributes 116

Drawing the Prototype 119

Defining Graphic Behaviors 122

Defining Interactive Behaviors 127

Editing a Prototype 128

Testing Your Prototype 129

Saving a Prototype 129

Loading and Saving Prototype Libraries **130**

Creating and Editing Prototype Instances in Panels **131**

Choosing a Buffer Type 131

Creating a Prototype Instance 131

Editing Prototype Instances 132

Loading and Saving Panels 132

Connecting Prototype Instances **132**

Chapter 8 The User Interface and Commands **134**

Overview **134**

Launching IBM ILOG Views Studio With the Prototypes Extension 135

The Main Window **135**

Buffer Windows 136

The Menu Bar 138
The Action Toolbar 139
The Editing Modes Toolbar 140
The Palettes Panel 140
Group Inspector Panel. 142
Prototypes Extension Commands. 143
CloseProtoLibrary 143
ConvertProtoManager 143
DeletePrototype 144
EditPrototype 144
GroupIntoGroup 144
NewProtoLibrary 145
NewPrototype. 145
NewPrototypeEditionBuffer 145
NewPrototypeGrapherBuffer 145
OpenProtoLibrary 146
SaveProtoLibraryAs 146
SelectGroupConnectionMode 146
SelectGroupSelectionMode 147
SelectNodeSelectionMode. 147
ShowGroupEditor 147
ToggleTimers 147
UngroupIlvGroups 148

Chapter 9 Using Prototypes in C++ Applications 150

Architecture 150
Groups 151
Attributes and Accessor Objects 152
Accessor Parameters 154
Prototypes and Instances 155
Displaying Groups and Instances in Managers and Containers 155
Connecting Attributes. 156

Linking Application Objects to Prototypes 156
Writing C++ Applications Using Prototypes 157
Header Files. 158
Loading a Panel Containing Prototype Instances 158
Loading Prototypes 159
Creating Prototype Instances. 160
Deleting Prototype Instances 160
Retrieving Groups and Prototype Instances 161
Getting and Setting Attributes 161
User-Defined and Predefined Attributes 163
Linking Prototypes to Application Objects 165
Setting Values Directly. 165
Using Group Mediators 166
Using Proto Mediators 168
Advanced Uses of Prototypes 169
Writing New Accessor Classes 169
Creating Prototypes by Coding 173
Customizing IBM ILOG Views Studio With the Prototypes Extension 175

Chapter 10 Predefined Accessors 178

Overview 178
Graphic Representation of the Behavior of a Prototype. 179
Data Accessors 180
Value 180
Reference 181
Group 181
Script 182
Control Accessors 184
Assign 184
Condition 185
Format 186
Increment 186

Min/Max 187
Multiple 188
Notify 189
Script 189
Switch 189
Toggle 191
Display Accessors 191
Fill 192
MultiRep 192
Rotation 193
ScaleX 194
ScaleY 195
TranslateX 195
TranslateY 196
Animation Accessors 197
Blink 197
Invert 198
Rotate 199
Trigger Accessors 200
Callback 200
Clock 201
Watch 201
Event 202
Miscellaneous Accessors 203
Debug 203
Prototype 204
Index 206

About This Manual

This manual describes the IBM® ILOG® Views 2D Graphics 5.3 products.

IBM ILOG Views 2D Graphics is used to develop very efficient 2D vector graphic representations with full interactive capabilities. It is composed of the Manager, Grapher, Prototype, and Web Deployment packages.

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

The manual contains four separate parts divided into chapters. Each of these parts describes one of the packages that make up IBM® ILOG® Views 2D Graphics 5.3, as follows:

- Part I, *Managers*, describes the IBM ILOG Views Managers package, dedicated to coordinating large quantities of graphic objects.
- Part II, *Grapher*, describes the IBM ILOG Views Grapher package, dedicated to creating graphic programs that include and represent hierarchical information.
- Part III, *Prototypes*, describes the IBM ILOG Views Prototypes package, dedicated to creating custom domain-specific graphic objects.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts, file names, and entries to be made by the user are written in `courier` typeface.

Naming Conventions

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

- The names of types, classes, functions, and macros defined in the IBM ILOG Views libraries 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:

```
virtual IlvClassInfo* getClassInfo() const;

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

Part I

Managers

Part I describes a high-level IBM® ILOG® Views package called the manager, which is dedicated to coordinating large quantities of graphic objects:

- Chapter 1, *Basic Manager Features* describes the classes, methods, and principles of managers.
- Chapter 2, *Manager Event Handling* describes the event handling mechanism of managers.
- Chapter 3, *Advanced Manager Features* describes the more advanced features of managers.

C H A P T E R

1

Basic Manager Features

This section describes how to coordinate a large quantity of graphic objects through the use of a *manager*, that is, through the `IlvManager` class and its associated classes.

The basic features of managers are described, in the following order:

- *Introducing Managers*
- *Manager Views*
- *Manager Layers*
- *Managing Objects*
- *Drawing and Redrawing*
- *Optimizing Drawing Tasks*
- *Saving and Reading*

Introducing Managers

A manager coordinates the interactions between the *display* of graphic objects in multiple views and the *organization* of graphic objects in multiple storage places. This is illustrated in Figure 1.1:

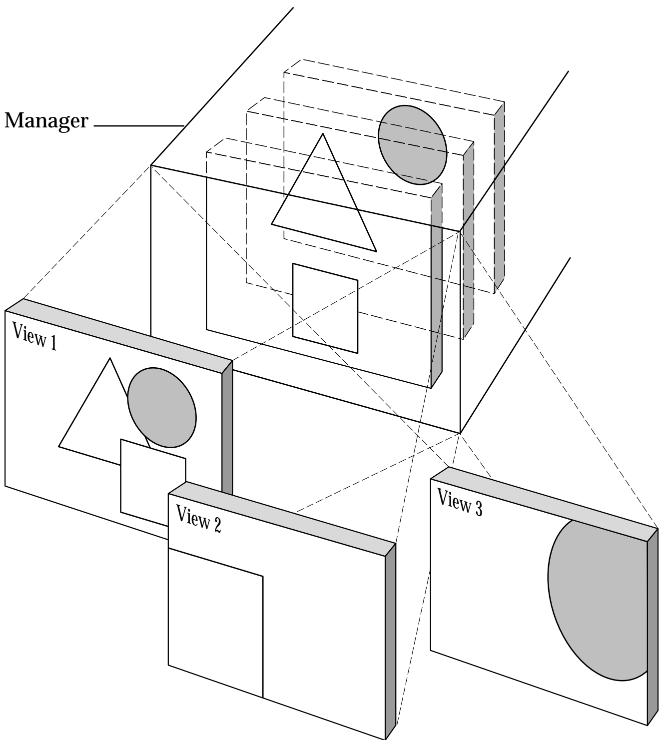


Figure 1.1 *Manager Concept*

To introduce some of the important concepts related to managers, the following items are described:

- *Layers*
- *Views*
- *View Transformer*
- *Event Handling*
- *Main Features of IlvManager*

Layers

Instances of the `IlvManager` class handle a set of graphic objects derived from the IBM® ILOG® Views class called `IlvGraphic`. When you organize graphic objects that the manager coordinates, you create an unlimited number of graphic objects and place them in multiple storage areas. These storage areas appear in superimposed layers. That is why they are called *manager layers*.

A manager is therefore a tool designed to handle objects placed in different *priority levels*. Priority level here means that objects stored in a higher screen layer are displayed in front of objects in lower layers.

Each graphic object stored in a layer is unique to that layer and can be stored only in that layer.

> ***Note:*** *An object must never be stored in more than one holder such as* `IlvManager`*,* `IlvContainer`*, or* `IlvGraphicSet`*.*

Graphic objects stored throughout the manager all share the same coordinate system.

Views

A manager uses one or multiple views to display its set of graphic objects. These views are instances of the class `IlvView` and you can connect as many as you want to the manager.

View Transformer

A geometric transformation (class `IlvTransformer`) can be associated with each view connected to a manager. When drawing its graphic objects in a view, the manager will use the transformer of the view, thereby providing a different representation of the same objects in each view (zoomed, unzoomed, translated, rotated, and so on).

Event Handling

All events are handled by means of event hooks, view interactors, object interactors, or accelerators. These are described briefly here and in more detail in section *Manager Event Handling*.

Event Hooks

The `IlvManagerEventHook` class is intended to monitor or filter events dispatched to the manager.

Interactors

Interactors are classes designed to handle user interactions involving a single or a complex combination of events.

- View interactors are classes derived from `IlvManagerViewInteractor` and handle interactions in the context of a whole view.
- Object interactors are derived from `IlvInteractor` and handle user interactions involving a single graphic object or a set of graphic objects.

Accelerators

An accelerator is an association of an event description with a user-defined action. In other words, when the event occurs the manager calls the action. This very basic interaction mechanism is limited to a single response to a single event, such as double-clicking with the left mouse button or pressing Ctrl-F.

Main Features of IlvManager

The `IlvContainer` class already provides ways of handling graphic objects. However, you may require more powerful features. Here is a list of circumstances under which you might need to use a manager:

- You need to handle a large number of graphic objects (hundreds or thousands) and encounter a performance problem using an `IlvContainer`.
- You wish to associate a specific behavior with a view, but not with a particular graphic object.
- You want multiple views of the same graphic objects, but without duplicating them. Remember that objects of the `IlvGraphic` class are not linked to any particular `IlvView`.
- You want to display the graphic objects with differing priorities.
- You want to add extra properties to objects, either individually or within a group, which would allow them to be visible or selectable.
- You want to save your graphic objects.

Managers provide a solution to these problems. They also offer advanced features that complex graphic applications may need:

- *Commands*
- *Input/Output*
- *Double-buffering*
- *Observers*
- *View Hooks*
- *Grid*

Commands

Objects can be manipulated and views can be changed by means of instances of the `IlvManagerCommand` class. This class has been designed to give `IlvManager` the ability to undo and redo changes.

Input/Output

Instances of the `IlvGraphic` class can deal with input/output. Similarly, the `IlvManager` class has a set of member functions to read and write object descriptions. Manager properties, such as the layer or name of an object, can also be read and written.

Double-buffering

When manipulating thousands of overlapping objects, redrawing operations can be very time-consuming. They can also be unattractive if each redrawn element reappears sequentially on the screen. These problems can be avoided by using the double-buffering technique implemented in `IlvManager`. When this feature is activated, all drawing functions are performed in a hidden image; when the area has been completely updated, the image is drawn at once in the working view.

Observers

This mechanism, based on the classes `IlvManagerObserver` and `IlvManagerObservable`, allows the application to be notified when certain modifications are done to the manager (adding or removing a view, setting a transformer on a view, adding graphic objects, adding or removing a layer, and so on).

View Hooks

Specific actions can be triggered under predefined circumstances. The manager view hooks let you connect events that occur in a manager with actions to be performed. This will be described in more detail in section *View Hooks*. Some application tasks performed with view hooks can be implemented with observers.

Grid

This tool allows you to force mouse events to occur only at locations defined by a snapping grid.

Manager Views

Attaching multiple views to a manager allows your program to display graphic objects simultaneously in various configurations. This is illustrated in Figure 1.2.

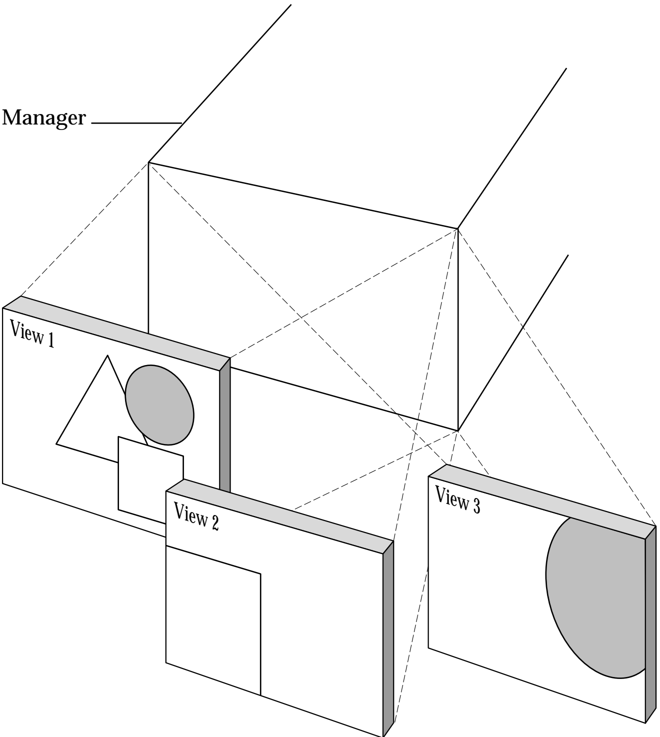


Figure 1.2 *Multiple Views Bound To a Manager*

The following `IlvManager` member functions handle the binding of views to a manager:

- `IlvManager::addView` - Attaches a view to the manager. All events are then handled by the hierarchy of interactors in place in the manager.
- `IlvManager::removeView` - Removes a view from the manager view list. The view is no longer handled by the manager.
- `IlvManager::getViews` - Returns an array of pointers to all the views currently connected to the manager.

The following aspects of manager views are described in this section:

- *View Transformations*
- *Double-buffering*

View Transformations

Use the following `IlvManager` member functions to modify the transformer associated with the view (except for `IlvManager::fitToContents`, which modifies the size of the view):

- `IlvManager::setTransformer`
- `IlvManager::addTransformer`
- `IlvManager::translateView`
- `IlvManager::zoomView`
- `IlvManager::rotateView`
- `IlvManager::fitToContents`
- `IlvManager::fitTransformerToContents`
- `IlvManager::ensureVisible`

Example: Zooming a View

This accelerator zooms a view using a scaling factor of two:

```
static void
ZoomView(IlvManager* manager, IlvView* view, IlvEvent& event, IlvAny)
{
   IlvPoint pt(event.x(), event.y());
   manager->zoomView(view, pt, IlvFloat(2), IlvFloat(2), IlvTrue);
}
```

The point given in the `zoomView` argument keeps its position after the zoom. The last parameter forces the redrawing of the view.

Double-buffering

The double-buffering member functions can be used to prevent the screen from flickering when many objects are manipulated. For each manager view, this feature requires the allocation of a hidden bitmap the size of the view. Depending on the number of views and the color model, double-buffering may consume a large amount of memory.

The member functions that handle double-buffering are:

- `IlvManager::isDoubleBuffering`
- `IlvManager::setDoubleBuffering`

- `IlvManager::setBackground`

Note: *You must use the* `setBackground` *member function to change the background color of a view in double-buffering mode.*

Example

This function switches the double-buffering mode of the given view:

```
static void
ToggleDoubleBuffering(IlvManager* manager, IlvView* view)
{
   manager->setDoubleBuffering(view,
                               !manager->isDoubleBuffering(view));
}
```

Manager Layers

Layers are storage places for graphic objects, as shown in Figure 1.3.

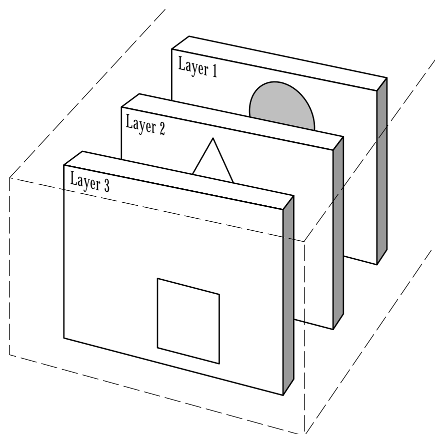


Figure 1.3 *Layers*

Once these objects have been stored they are controlled by and organized under the same manager. Each layer is unique to and can be controlled by only one manager. Each graphic object handled by a manager belongs to one and only one layer.

Note: *For more member functions dealing with layers, see the* `IlvManager` *and* `IlvManagerLayer` *classes.*

This section is divided as follows:

- *Layer Index*
- *Layer Selectability*
- *Layer Visibility*
- Layer Rendering

Layer Index

Layers are stored by the manager according to their index. The first layer has index `0` and layer `N` has index `N-1`. Layers are represented by an instance of the `IlvManagerLayer` class, but most of the time they are identified in member function signatures by their index in the manager. Various member functions let you manipulate these layers or the objects that they own.

The manager draws the layers one by one, starting at index `0`. Consequently, the top-most layer on the screen is the one with the highest index. This introduces a visual hierarchy among graphic objects based on their layer index. In general, graphic objects of a more static nature—for instance, objects used as background for your IBM ILOG Views programs—are put in a lower layer of the manager. Graphic objects of a dynamic nature—objects with which users interact—are typically put in a higher layer. The top-most layer (the one with the highest index) is reserved for use by the manager; it contains the selection objects displayed as square handles around selected objects. Since the manager increases the index of this layer as new layers are added, it always remains on the top of the stack.

Setting-Up Layers

By default, a manager is created with two layers. You can change this number when creating a manager by using the second parameter of the constructor. You can also change this number once the manager has been created, by using the `IlvManager::setNumLayers` member function.

> ***Reminder:*** *You must refer to the layers by index numbers starting with 0. For example, layer 3 is indexed as 2.*

Example

The following code adds an object to the second layer (specified by index `1`) of the manager and then moves the object to layer `0`.

```
manager->addObject(object, IlvTrue, 1);
manager->setLayer(object, 0);
```

When adding a graphic object using a non-existing layer index, the number of layers is increased automatically.

```
IlvManager* manager = new IlvManager(display); // A manager with 2 layers
IlvRectangle* rect = new IlvRectangle(display, IlvRect(0, 0, 100, 100));
// Add the object in layer 7 and create intermediate layers
manager->addObject(rect, IlFalse, 7);
```

Layer Selectability

Layer selectability indicates whether the application end-user can select the objects within a certain layer. Preventing your program user from selecting graphic objects in a layer means that these objects are fixed and unchangeable. The following member functions are used for layer selectability:

- IlvManager::setSelectable
- IlvManager::isSelectable

Layer Visibility

Layer visibility indicates whether the objects within a certain layer should be visible to the user. This notion of layer visibility is not as simple as it seems because a layer can be hidden in several different ways:

- Globally - Hidden in all the manager views.
- Locally - Hidden in one or several manager views.
- Contextually - Hidden by an application visibility filter.

A layer is displayed in a view if it is not hidden in any of these ways.

Global Visibility

If a layer is hidden globally, it will not be displayed in any of the manager views. The following `IlvManager` member functions allow you to get or set the global visibility of a layer:

- `setVisible (int layer, IlBoolean val)`
- `isVisible (int layer)`

Local Visibility

Use the following `IlvManager` member functions to get or set the visibility of a layer for a given manager view:

- `setVisible (const IlvView* view, int layer, IlBoolean visible)`
- `isVisible (const IlvView* view, int layer)`

Visibility Filter

`IlvLayerVisibilityFilter` is an abstract class. Subclasses must redefine the virtual member function `IlvLayerVisibilityFilter::isVisible` to return the visibility status of the layer.

Each manager layer handles a list of visibility filters. When a layer must be drawn in a view, the manager calls the member function `IlvLayerVisibilityFilter::isVisible` for all the filters of the layer; if a visibility filter returns `IlFalse`, the layer is not displayed. This mechanism only allows the application to hide layers that would be otherwise visible; it does not allow you to show hidden layers.

To add a visibility filter to a layer, use `IlvManagerLayer::addVisibilityFilter`.

Layer Rendering

Layer rendering indicates how the layer is to be rendered onto the drawing device. Two attributes of the layer can change its rendering:

- *Alpha Value*
- *Anti-aliasing Mode*

Alpha Value

The alpha value of a layer represents the opacity with which this layer will be drawn above other layers. If the layer contains objects having transparent colors, the transparency of the layer and the transparent objects will be composed.

The default value for this setting is `IlvFullIntensity`, which means that the layer is completely opaque.

See the `IlvManagerLayer::setAlpha` method for details.

Anti-aliasing Mode

The anti-aliasing mode of a layer is a global setting that will be applied to all the objects of this layer. It indicates the anti-aliasing mode with which objects are going to be rendered.

The default value for this setting is `IlvDefaultAntialiasingMode`, which means that the anti-aliasing mode of the layer will be inherited from the drawing port itself. For example, if the anti-aliasing mode of a manager view has been set to `IlvUseAntialiasingMode` (see `IlvPort::setAntialiasingMode`), it means that all the layers of this view will use anti-aliasing. You can override this setting for a specific layer by indicating that you do not need anti-aliasing for this layer.

See the `IlvManagerLayer::setAntialiasingMode` method for details.

> ***Note:*** *These features are only supported on Microsoft Windows with GDI+ installed. See Appendix B / GDI+ of the Foundation User's Manual for details*

Managing Objects

This section explains how to manipulate the objects contained in a manager. It is divided as follows:

- *Modifying the Geometry of Graphic Objects*
- *Selecting Objects*
- *Selection Procedures*
- *Managing Selected Objects*
- *Managing Object Properties*
- *Arranging Objects*

Modifying the Geometry of Graphic Objects

The `IlvManager` class has been designed to handle a large number of graphic objects. In order to perform graphical operations efficiently (for example, redrawing part of a view, locating the objects at a given position, and so on), the manager uses a complex internal data structure where graphic objects are organized according to their geometry, that is, their bounding box. To keep this data structure up to date, the manager needs to be aware of any modification in the geometry of its graphic objects. This is why any such modification should be carried out in the following manner:

1. Take the object out of the manager list.
2. Manipulate its geometric characteristics.
3. Put the object back into the manager list.

The easiest way to do this is to use the dedicated `IlvManager` member functions respecting these requirements:

- `IlvManager::applyToObject`
- `IlvManager::applyToObjects`
- `IlvManager::applyInside`
- `IlvManager::applyIntersects`
- `IlvManager::applyToTaggedObjects`
- `IlvManager::applyToSelections`

> ***Note:*** *Do not change the size of a managed object by calling its* `IlvGraphic::translate` *or* `IlvGraphic::scale` *member functions. The manager use sophisticated data structures and an intricate indexing system for tracking the position of objects with respect to each other. You should not interfere with these mechanisms.*

For simple geometric operations such as moving, translating, or reshaping, `IlvManager` provides the following member functions that do not need to call `IlvManager::applyToObject`:

- `IlvManager::translateObject`
- `IlvManager::moveObject`
- `IlvManager::reshapeObject`

Example: Translating an Object

The following code gets a pointer to an object named `test` from the manager. If this object exists, it is translated `10` pixels right and `20` pixels down, and then redrawn (fourth parameter set to `IlTrue`):

```
object = manager->getObject("test");
if (object)
   manager->translateObject(object, 10, 20, IlvTrue);
```

Applying Functions to Objects in a Region

In order to apply a user-defined function to objects that are located either partly or wholly within a specific region, use the following `IlvManager` member functions:

- `IlvManager::applyInside`
- `IlvManager::applyIntersects`

Selecting Objects

Use the following two member functions of `IlvManager` to handle the selection state of objects:

- `IlvManager::isSelected`
- `IlvManager::setSelected`

Example:

The following code gets a pointer to an object named `test` from the manager. If this object exists, it is selected (second parameter is set to `IlTrue`) and redrawn (third parameter set to `IlTrue`):

```
object = manager->getObject("test");
if (object)
   manager->setSelected(object, IlvTrue, IlvTrue);
```

Selection Procedures

The `IlvManager` member functions involved in selection tasks are the following:

- IlvManager::applyToSelections
- `IlvManager::numberOfSelections`
- `IlvManager::deSelectAll`
- `IlvManager::getSelections`
- `IlvManager::deleteSelections`
- `IlvManager::getSelection`
- `IlvManager::setMakeSelection`

Example: Customizing Selection Handle Objects

This example shows how to attach new selection handle objects to line objects:

```
static IlvDrawSelection*
MakeSelection(IlvManager* manager, IlvGraphic* graphic)
{
   if (graphic->isSubtypeOf("IlvLine"))
      return new IlvLineHandle(manager->getDisplay(), graphic);
   else
      return new IlvDrawSelection(manager->getDisplay(), graphic);
}
```

The following code changes the function called to create the selection object. If the selected object is an `IlvLine` or an instance of a class derived from it, the manager uses the `IlvLineHandle` object to draw the selection:

```
manager->setMakeSelection(MakeSelection);
```

Managing Selected Objects

Selecting is a basic process for managers and most manager functions should apply to a selected list of objects. A manager selection can be thought of as a special set holding some of the managed objects. To display selected objects within a manager, IBM® ILOG® Views

creates *selection objects* that are stored in the manager. The difference between these objects and others is that they are internally managed and cannot be manipulated.

Example: Translating the Selected Objects

The following example shows an accelerator that translates all selected objects ten pixels right and 20 pixels down. This accelerator uses the `IlvManager::applyToSelections` member function to translate each of the objects. Redrawing of the objects is done once at the end of the call to this method, as is done for all the apply functions, because its third parameter is set to the default value `IlTrue`.

```
static void
TranslateSelectedObjects (IlvGraphic* object, IlvAny arg)
{
   IlvManager* manager = (IlvManager*) arg;
   manager->translateObject(object, 10, 20, IlvFalse);
}

static void
TranslateAccelerator(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   manager->applyToSelections(TranslateSelectedObjects, manager);
}
```

Managing Object Properties

Several member functions of the `IlvManager` class describe properties that are assigned to an object when it is added to a manager (for example, `IlvManager::isSelectable`, `IlvManager::setSelectable`, `IlvManager::isResizeable`, and so on).

You can also add specific properties to each object by means of the property-related member functions of the `IlvGraphic` class. These properties are application-dependent and have no effect on the manager.

`IlvManager` provides member functions to check whether an object has a property or to change a property of an object.

Example: Setting an Object as Unmovable

This is an example of how to set an object in a manager as unmovable:

```
object = manager->getObject("test");
if (object)
   manager->setMoveable(object, IlvFalse);
```

Arranging Objects

The `IlvManager` class provides member functions to help organize the layout of graphic objects.

- *Grouping*
- *Aligning and Duplicating*

Grouping

The `IlvManager::group` member function lets you create an `IlvGraphicSet` from an array of objects and put the objects from an `IlvGraphicSet` into the manager.

The `IlvManager::unGroup` member function lets you do the inverse of this.

> ***Note:*** *Graphic objects grouped in a graphic set are no longer handled by the manager. The manager only sees the graphic set.*

Example: Grouping Objects

This is an example of an accelerator that groups selected objects:

```
static void
Group(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   if (!manager->numberOfSelections()) return;
   IlvUInt n;
   IlvGraphic* const* objs = manager->getSelections(n);
   IlvGraphicSet* g = manager->group(n, (IlvGraphic* const*)objs);
   if (g) manager->setSelected((IlvGraphic*)g, IlvTrue, IlvTrue);
}
```

The first line checks the number of objects and returns if no objects are selected. Then, a pointer to the selected objects is obtained using the `IlvManager::getSelections` member function. The next line creates the group. The new object is selected at the end of this accelerator.

Aligning and Duplicating

Some `IlvManager` member functions are defined to automatically align objects with respect to each other:

- `IlvManager::align`
- `IlvManager::makeColumn`
- `IlvManager::makeRow`
- `IlvManager::sameWidth`
- `IlvManager::sameHeight`

Another member function duplicates objects, that is, it creates a copy of the objects and inserts them into the manager:

- `IlvManager::duplicate`

> ***Note:*** *These modifications are always applied to the currently selected objects*

Example: Make All Selected Objects the Same Width

This accelerator gives the width of the first selected object to all the selected objects:

```
static void
SameWidth(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   manager->sameWidth(IlvTrue);
}
```

The value `IlTrue` passed to `IlvManager::sameWidth` indicates that the objects are automatically redrawn.

Drawing and Redrawing

Use the following `IlvManager` member functions to draw objects:

- `IlvManager::draw`
- `IlvManager::reDraw`
- `IlvManager::bufferedDraw`

The `IlvManager::bufferedDraw` method works in the same way as double-buffering does, with the following differences:

- It is local to a view, a region, or an object.
- It only lasts for the duration of the drawing operation.

The next section, *Optimizing Drawing Tasks*, describes other `IlvManager` member functions used to redraw graphic objects efficiently in a manager.

Redrawing All Views

In some cases, you may want to refresh all the views managed by an `IlvManager`. To do so, call one of the `IlvManager::reDraw` member functions:

```
manager->reDraw();
```

Optimizing Drawing Tasks

A special manager feature lets you perform several geometric operations and redraw only when all the modifications are done. This is implemented by the use of the *update region*, which is a region made up of invalidated rectangles.

The update region stores the appropriate regions before any modifications are carried out on objects. It also stores the relevant regions after these modifications have been carried out for each view.

To successfully perform an application task, you must mark the regions where relevant objects are located as invalid, apply the function, and then invalidate the regions where the objects involved are now placed. This mechanism is simplified by means of a set of member functions of the `IlvManager` class. Regions to be updated are refreshed only when `IlvManager::reDrawViews` is called, which means that refreshing the views of a manager is done by marking regions to be redrawn in a cycle of `IlvManager::initReDraws` and `IlvManager::reDrawViews`.

These cycles can be nested so that only the last call to the `IlvManager::reDrawViews` member function actually updates the display.

The `IlvManager` member functions that help you optimize drawing tasks are:

- `IlvManager::initReDraws` - Marks the beginning of the drawing optimization operation by emptying the region to update for each managed view. Once this step is completed, direct or indirect calls to a draw directive are deferred. For every `IlvManager::initReDraws`, there should be one call to `IlvManager::reDrawViews`, or else a warning is issued. Calls to `IlvManager::initReDraws` can be embedded so that the actual refresh takes place only when the last call to `IlvManager::reDrawViews` is reached.
- `IlvManager::invalidateRegion` - Marks a region as invalid. This region will be redrawn later. Each call to `IlvManager::invalidateRegion` adds the region to the update region in every view.
- `IlvManager::reDrawViews` - Sends the drawing commands for the whole update region. All the objects involved in previous calls to `IlvManager::invalidateRegion` are then updated.
- `IlvManager::abortReDraws` - Aborts the mechanism of deferred redraws (for example, if you need to refresh the whole screen). This function resets the update region to empty. If needed, you should start again with an `IlvManager::initReDraws` call.
- `IlvManager::isInvalidating` - Returns `IlTrue` when the manager is in an `IlvManager::initReDraws/IlvManager::reDrawViews` state.

The successive use of these member functions is a mechanism used in the `IlvManager::applyToObject` member function. In fact, the call:

```
manager->applyToObject(obj, func, userArg, IlvTrue);
```

is equivalent to:

```
manager->initReDraws();
manager->invalidateRegion(obj);
manager->applyToObject(obj, func, userArg, IlvFalse);
manager->invalidateRegion(obj);
manager->reDrawViews();
```

The `IlvManager::invalidateRegion` member function works with the bounding box of the object given in the parameter. When an operation applied to the object modifies its bounding box, `IlvManager::invalidateRegion` must be called twice; once before and once after the operation.

For example, when moving an object, you must invalidate the region where the object was initially located and invalidate the final region so that the object can be redrawn. If the object bounding box is not modified, only one call to `IlvManager::invalidateRegion` is necessary.

Saving and Reading

Manager objects and their properties can be saved and read from particular streams. To make it easy to save and restore a set of `IlvGraphic` objects, two classes are provided:

- `IlvManagerOutputFile` (a subtype of `IlvOutputFile`)
- `IlvManagerInputFile` (a subtype of `IlvInputFile`)

These two classes add only manager-specific information to the object description blocks.

The `IlvManagerInputFile` class reads the files that have been created using `IlvManagerOutputFile`.

Example: Using the IlvManagerOutputFile Class

The following is an example of subtyping of the `IlvOutputFile` class, where the `IlvOutputFile::writeObject` member function is implemented to add the manager-specific information for each object:

```
void
IlvManagerOutputFile::writeObject(const IlvGraphic* object)
{
    if (getManager()->isManaged(object))
        getStream() << getManager()->getLayer(object) << IlvSpc();
    else
        getStream() << "-1 ";
    writeObjectBlock(object);
}
```

New information is added before the object descriptor block is written. It indicates the layer where the graphic `object` lies. If the object was not managed by the manager, IBM ILOG Views writes the value `-1` to `getStream` (which is not a valid layer index). The value `-1` indicates that the object should not be added to the manager object set.

> ***Note:*** *Specialized IBM ILOG Views graphic objects called "gadgets" need the following subclasses:* `IlvGadgetManagerInputFile` *(subclass of* `IlvInputFile`*) and* `IlvGadgetManagerOutputFile` *(subclass of* `IlvOutputFile`*). These subclasses handle the persistence of gadget-related properties. Subtyping these two classes is allowed, but it is mandatory to insert the string* `"Gadget"` *in the subtyped C++ class name.*

The C++ code used to implement the `IlvManagerInputFile::readObject` member function is shown here:

```
IlvGraphic*
IlvManagerInputFile::readObject()
{
    IlvGraphic* object;
    int layer;
    getStream() >> layer;
    IlUInt dummyIndex;
    IlvGraphic* object = readObjectBlock(dummyIndex);
    if (object && (layer >= 0))
        getManager()->addObject(object, IlFalse, layer);
    return object;
}
```

The object read is added to the manager only if its layer index is greater than or equal to `0`.

CHAPTER

2

Manager Event Handling

This section describes how managers handle events.

An event can be handled by different types of manager components:

- *Event Hooks*
- *View Interactors*
- *Object Interactors*
- *Accelerators*

First, the mechanism for handling events is described. Then, the different manager components that handle events are presented.

The Event Handling Mechanism

The mechanism used by a manager when it receives an event is as follows:

1. It sends the event to the list of event hooks.
2. If none of the event hooks consume the event, it is sent to the interactor associated with the view that received the event.

3. If there is no view interactor, the manager looks for the top most graphic object at the event position and sends the event to its object interactor.
4. If there is no object or no object interactor, or if the object interactor does not handle the event, it is dispatched to the manager accelerators.

Event Hooks

Event hooks are instances of the `IlvManagerEventHook` class. They are used to monitor or filter events occurring in all the views associated with the manager. Each manager has a list of event hooks. They can be added or removed from the list using the following `IlvManager` member functions:

- `IlvManager::installEventHook`
- `IlvManager::removeEventHook`

Event hooks are the first ones to get hold of the events occurring in a manager.

When it receives an event, the manager calls the `handleEvent` member function of each event hook one after the other. If one of them returns `IlTrue`, the subsequent event hooks are not called and the event is considered to be consumed. If none of the event hooks consume the event, it is dispatched further to interactors or accelerators.

View Interactors

The role of the `IlvManagerViewInteractor` class is to handle complex sequences of user events to be treated by a particular `IlvView` associated with a manager.

Setting or removing an interactor on a view can be done using the following `IlvManager` member functions:

- `IlvManager::getInteractor`
- `IlvManager::setInteractor`
- `IlvManager::removeInteractor`

In this section, the predefined view interactors are first listed and then two examples showing how to implement view interactors are presented, as follows:

- *Predefined View Interactors*
- *Example: Implementing the IlvDragRectangleInteractor Class*
- *Example of an Extension: IlvMoveInteractor*

Predefined View Interactors

Predefined interactors obtained by instantiating subclasses derived from the `IlvDragRectangleInteractor` class are listed here:

- `IlvDragRectangleInteractor`

  Lets the user draw a rectangle that can be used for any purpose by subclasses (see section *Example: Implementing the IlvDragRectangleInteractor Class* for an example showing how to use this interactor).

  Include `<ilviews/manager/dragrin.h>`

- `IlvMakeRectangleInteractor`

  Allows you to create `IlvRectangle` objects.

  Include `<ilviews/manager/mkrectin.h>`

- `IlvMakeFilledRectangleInteractor`

  Allows you to create `IlvFilledRectangle` objects.

  Include `<ilviews/manager/mkrectin.h>`

- `IlvMakeReliefRectangleInteractor`

  Allows you to create `IlvReliefRectangle` objects.

  Include `<ilviews/manager/mkrelfin.h>`

- `IlvMakeReliefDiamondInteractor`

  Allows you to create `IlvReliefDiamond` objects.

  Include `<ilviews/manager/mkrelfin.h>`

- `IlvMakeRoundRectangleInteractor`

  Allows you to create `IlvRoundRectangle` objects.

  Include `<ilviews/manager/mkround.h>`.

- `IlvMakeFilledRoundRectangleInteractor`

  Allows you to create `IlvFilledRoundRectangle` objects.

  Include `<ilviews/manager/mkround.h>`

- `IlvMakeEllipseInteractor`

  Allows you to create `IlvEllipse` objects.

  Include `<ilviews/manager/mkarcin.h>`

- `IlvMakeFilledEllipseInteractor`

Allows you to create `IlvFilledEllipse` objects.

Include `<ilviews/manager/mkarcin.h>`

- `IlvMakeZoomInteractor`

  Handles the zooming command. You draw a rectangular region into which you wish to zoom.

  Include `<ilviews/manager/geointer.h>`

- `IlvMakeUnZoomInteractor`

  Handles the unzooming command. You draw a rectangular region into which the area you are watching is unzoomed.

  Include `<ilviews/manager/geointer.h>`

- `IlvMakeBitmapInteractor`

  Allows you to create a bitmap from the view. You drag a rectangle and an `IlvIcon` object is created from the contents of the rectangle selected.

  Include `<ilviews/manager/utilint.h>`

- `IlvSelectInteractor`

  Allows you to select, translate, and resize graphic objects.

  Include `<ilviews/manager/selinter.h>`

- `IlvMakeLineInteractor`

  Allows you to create `IlvLine` objects. Two derived classes are defined to create different types of lines: `IlvMakeArrowLineInteractor` and `IlvMakeReliefLineInteractor`.

  Include `<ilviews/manager/mklinein.h>`

Example: Implementing the IlvDragRectangleInteractor Class

This example demonstrates how the `IlvDragRectangleInteractor` member functions are implemented. The example can be used as a starting point to create your own interactor.

The `IlvDragRectangleInteractor` interactor allows the user to designate a rectangular region in a view. This rectangle can then be used for various purposes in derived interactors; for instance, a subclass dedicated to the creation of a graphic object can use the rectangle to define the bounding box of the new object.

Here is a slightly revised version of the synopsis of this class:

```
class IlvDragRectangleInteractor
: public IlvManagerViewInteractor
{
public:
    IlvDragRectangleInteractor(IlvManager* manager, IlvView* view)
       : IlvManagerViewInteractor(manager, view) {}

    virtual void handleEvent(IlvEvent& event);
    virtual void drawGhost();
    virtual void doIt(IlvRect&);
    virtual void abort();

    IlvRect& getRectangle();
protected:
    IlvRect _xor_rectangle;
    IlvPos  _firstx;
    IlvPos  _firsty;
};
```

Three protected fields are defined:

- `_xor_rectangle` - Holds the coordinates of the rectangle being dragged by the user.
- `_firstx` and `_firsty` - The coordinates of the first button-down event received. This point is used as the start of the selected rectangle. It can be any one of the 4 corners depending on the direction in which the user drags the rectangle.

The constructor does nothing and the initialization is done by the `doIt` member function.

Also, four member functions of the `IlvManagerViewInteractor` class are overloaded:

- *abort Member Function*
- *handleEvent Member Function*
- *drawGhost Member Function*
- *doIt Member Function*

abort Member Function

This member function is called to cancel the interaction. The rectangle width is set to `0`.

```
void
IlvDragRectangleInteractor::abort()
{
    _xor_rectangle.w(0);
}
```

handleEvent Member Function

The following shows a simplified version of the `IlvDragRectangleInteractor::handleEvent` member function.

```
void
```

```
IlvDragRectangleInteractor::handleEvent(IlvEvent& event)
{
    switch(event.type()) {
    case IlvKeyUp:
    case IlvKeyDown:
        getManager()->shortCut(event, getView());
        break;
    case IlvButtonDown:
        if (event.button() != IlvLeftButton)
            getManager()->shortCut(event, getView());
        else {
            _xor_rectangle.w(0);
            IlvPoint p(event.x(), event.y());
            if (getTransformer()) getTransformer()->inverse(p);
            _firstx = p.x();
            _firsty = p.y();
        }
        break;
   case IlvButtonDragged:
        if ((event.button()    != IlvLeftButton))
            getManager()->shortCut(event, getView());
        else {
            if (_xor_rectangle.w()) drawGhost();
            IlvPoint p(event.x(), event.y());
            if (getTransformer()) getTransformer()->inverse(p);
            _xor_rectangle.move(IlvMin(_firstx, p.x()),
                                IlvMin(_firsty, p.y()));
            _xor_rectangle.resize((IlvDim)(IlvMax(_firstx, p.x())
                                                 -_xor_rectangle.x()),
                                  (IlvDim)(IlvMax(_firsty, p.y())
                                                -_xor_rectangle.y()));
            ensureVisible(IlvPoint(event.x(), event.y()));
            drawGhost();
        }
        break;
     case IlvButtonUp:
        if (event.button() != IlvLeftButton)
            getManager()->shortCut(event, getView());
        else {
            if (!_xor_rectangle.w()) return;
            drawGhost();
            IlvRect rect(_xor_rectangle);
            _xor_rectangle.w(0);
            doIt(rect);
        }
        break;
    }
```

Here, only button events are managed. Other events are discarded or sent to the manager for possible dispatch to accelerators by means of a call to the `IlvManager::shortCut` member function.

The following types of events are handled by the `handleEvent` member function:

- *Keyboard Events*
- *Button-Down Events*

- *Button-Dragged Events*
- *Button-Up Events*

Keyboard Events

You want to ignore these events. The best way to do this without losing the information conveyed by the event is to bypass the natural view interactor process and send the event back to the manager where it might match an accelerator:

```
case IlvKeyUp:
case IlvKeyDown:
        getManager()->shortCut(event, getView());
        break;
```

Button-Down Events

```
case IlvButtonDown:
...
break;
```

The mouse position is stored in `_firstx` and `_firsty` and the rectangle is reset. This is done by setting the width of the rectangle to `0`. Then, the coordinates are stored in the object coordinate system:

```
if (event.button() != IlvLeftButton)
   getManager()->shortCut(event, getView());
else {
   _xor_rectangle.w(0);
   IlvPoint p(event.x(), event.y());
   if (getTransformer()) getTransformer()->inverse(p);
   _firstx = p.x();
   _firsty = p.y();
}
```

Button-Dragged Events

```
case IlvButtonDragged:
...
break;
```

If `_xor_rectangle` is valid, the rectangle has been drawn with `drawGhost` and has to be erased:

```
if (_xor_rectangle.w()) drawGhost();
```

The new rectangle is computed in the object coordinate system:

```
IlvPoint p(event.x(), event.y());
if (getTransformer()) getTransformer()->inverse(p);
_xor_rectangle.move(IlvMin(_firstx, p.x()),
                    IlvMin(_firsty, p.y()));
_xor_rectangle.resize((IlvDim)(IlvMax(_firstx, p.x())
                               -_xor_rectangle.x()),
                      (IlvDim)(IlvMax(_firsty, p.y())
                               -_xor_rectangle.y()));
```

The following ensures that the dragged point remains on the screen. When the view is in a scrolled view, you can change the view coordinates to keep the mouse position visible:

```
ensureVisible(IlvPoint(event.x(), event.y()));
```

The new rectangle is drawn:

```
drawGhost();
```

Button-Up Events

A button-up event signifies the end of the interaction; the rectangle has been defined:

```
case IlvButtonUp:
...
break;
```

The previous ghost image is erased:

```
drawGhost();
```

The current rectangle is saved and the interactor is reset:

```
IlvRect rect(_xor_rectangle);
_xor_rectangle.w(0);
```

The `doIt` virtual member function is called. Subclasses overload this method to perform their final task using the rectangle provided as the parameter:

```
doIt(rect);
```

drawGhost Member Function

The `IlvDragRectangleInteractor::drawGhost` member function draws a ghost image of `_xor_rectangle`:

```
void
IlvDragRectangleInteractor::drawGhost()
{
    IlvManager* mgr = getManager();
    if (_xor_rectangle.w()) {
        IlvRect rect = _xor_rectangle;
        if(getTransformer()) getTransformer()->apply(rect);

        getView()->drawRectangle(mgr->getPalette(),rect);
    }
}
```

Because `_xor_rectangle` is expressed in the object coordinate system, the transformer of the view must be applied before drawing the rectangle.

doIt Member Function

The `IlvDragRectangleInteractor::doIt` member function does nothing; it is designed to be overloaded to perform actions once the user has selected a rectangular region.

Two examples of how to overload this member function are presented:

- The first example shows how to create a new `IlvRectangle` object with the rectangular region (the same way as the `IlvMakeRectangleInteractor` class does).
- The second example shows how to select all the objects located in the rectangular region. This illustrates how to manipulate the selection within a manager without using the select interactor.

Example 1: IlvMakeRectangleInteractor

Here is a simplified version of the `IlvMakeRectangleInteractor::doIt` member function, derived from the `IlvDragRectangleInteractor` class. This member function deselects all the objects of the manager, creates an `IlvRectangle` instance, adds it to the manager, and sets the selection on it.

```
void
IlvMakeRectangleInteractor::doIt(IlvRect& rect)
{
    IlvGraphic* obj = new IlvRectangle(getDisplay(), rect);
    getManager()->deSelect();
    getManager()->addObject(obj);
    getManager()->makeSelected(obj);
}
IlvGraphic* obj = new IlvRectangle(getDisplay(), rect);
```

Example 2: Selector

This example shows how to implement a simple interactor to select graphic objects. The `IlvDragRectangleInteractor::doIt` member function is overloaded in order to select every object located within the region the user has created.

The `SelectAnObject` function is defined. This is called by an application member function of the manager. The manager is available in the `manager` parameter:

```
static void
SelectAnObject(IlvGraphic* object, IlvAny manager)
{
    ((IlvManager*)manager)->setSelected(object, IlTrue);
}
```

The `doIt` member function calls `SelectAnObject` for each object located in the designated rectangle. To find these objects, call the manager member function `applyInside`:

```
void
MyRectangleSelector::doIt(IlvRect& rect)
{
    getManager()->applyInside(rect, SelectAnObject, (IlvAny)getManager());
}
```

Example of an Extension: IlvMoveInteractor

This is a complete example of a direct subtype of the `IlvManagerViewInteractor` class. It allows the user to move a graphic object to another location by dragging it with the mouse. Here is the declaration of this class (it can also be found in the header file `<ilviews/manager/movinter.h>`):

```
class IlvMoveInteractor
: public IlvManagerViewInteractor
{
public:
    IlvMoveInteractor(IlvManager* manager,
                      IlvView*    view)
        : IlvManagerViewInteractor(manager, view),
          _move(0) {}

    virtual void   handleEvent(IlvEvent& event);
    virtual void   handleExpose(IlvRegion* clip = 0);
    virtual void   drawGhost();
    void           drawGhost(const IlvRect&,
                             IlvRegion* clip = 0);
    void           drawGhost(IlvGraphic*, IlvRegion* clip = 0);
    virtual void   doIt(const IlvPoint&);
    const IlvRect& getRectangle() const {return _xor_rectangle;}
protected:
    IlvPos         _deltax, _deltay;
    IlvRect        _bbox;
    IlvGraphic*    _move;
    IlvRect        _xor_rectangle;
    IlBoolean     _wasSelected;
    void           handleButtonDown(const IlvPoint&);
    void           handleButtonDragged(const IlvPoint&);
    void           handleButtonUp(const IlvPoint&);
};
```

This interactor lets you select and deselect objects by clicking on them with the left mouse button and the Shift key pressed. You can move an object or a set of selected objects but you cannot resize them.

The following protected fields are used in this class:

- `_deltax, _deltay` - Stores the distance between the mouse and the top-left corner of the objects being moved.
- `_bbox` - Stores the bounding box of the objects being moved.
- `_move` - Keeps a pointer to the object being moved.
- `_xor_rectangle` - Stores the rectangle dragged to mark a region.
- `_wasSelected` - Keeps a Boolean value indicating whether the designated object was selected before it was moved. This information is required because the object is selected when you start to move it. There are two different cases in this interactor, depending on

whether one or more object is being moved. If more than one object is moved, a moving rectangle that encloses the bounding boxes of these objects is displayed. Otherwise, the moving objects themselves are displayed.

The following member function are described in this section:

- *handleEvent Member Function*
- *drawGhost Member Function*
- *drawGhost for a Rectangle*
- *drawGhost for an Object*
- *doIt Member Function*
- *handleButtonDown Member Function*
- *handleButtonDragged Member Function*
- *handleButtonUp Member Function*

handleEvent Member Function

The following code focuses on mouse events. All other events are dispatched to accelerators by a call to `IlvManager::shortCut`, but only if an object is not being moved at this point. This is because some accelerators might remove the object being worked on, which can be dangerous:

```
void
IlvMoveInteractor::handleEvent(IlvEvent& event)
{
    switch (event.type()) {
    case IlvButtonDown:
        _xor_rectangle.w(0);
        _move = 0;
        if (event.modifiers() & (IlvLockModifier | IlvNumModifier)) {
            getManager()->getDisplay()->bell();
            return;
        }
        if (event.button() != IlvLeftButton) {
            getManager()->shortCut(event, getView());
            return;
        }
        if (!event.modifiers())
            handleButtonDown(IlvPoint(event.x(), event.y()));
        else {
            IlvManager* manager = getManager();
            if (event.modifiers() & IlvShiftModifier) {
                IlvPoint p(event.x(), event.y());
                IlvGraphic* obj = manager->lastContains (p,getView());
                IlvDrawSelection* sel = 0;
                if (obj) sel = getSelection(obj);
                if (!sel && obj && manager()->isSelectable(obj)) {
                    manager->setSelected(!manager->isSelected(obj));
                }
            } else
```

```
                manager->shortCut(event, getView());
        }
        break;
    case IlvButtonUp:
        if (event.button() == IlvLeftButton)
            handleButtonUp(IlvPoint(event.x(), event.y()));
        else getManager()->shortCut(event, getView());
        break;
    case IlvButtonDragged:
        if (event.modifiers() == IlvLeftButton){
            IlvPoint p(event.x(), event.y());
            handleButtonDragged(p);
        }
        break;
    default:
        if (!_move)
            getManager()->shortCut(event, getView());
        break;
    }
```

The following types of events are handled by the `handleEvent` member function:

- *Button-Down Events*
- *Button-Up Events*
- *Button-Dragged Events*

Button-Down Events

The interactor is initialized by setting `_move` and `_xor_rectangle`:

```
_xor_rectangle.w(0);
_move = 0;
```

Only the left button is handled. If the event involves another mouse button, the event is ignored and dispatched to manager accelerators:

```
if (event.button() != IlvLeftButton) {
    getManager()->shortCut(event, getView());
    return;
}
```

The `handleButtonDown` member function is called if there is no event modifier:

```
if (!event.modifiers())
    handleButtonDown(IlvPoint(event.x(), event.y()));
```

If the Shift modifier is set, the selection state of the object pointed to by the mouse is toggled:

```
if (event.modifiers() & IlvShiftModifier) {
    IlvPoint p(event.x(), event.y());
    IlvGraphic* obj = manager->lastContains(p, getView());
    IlvDrawSelection* sel = 0;
    if (obj) sel = getSelection(obj);
    if (!sel && obj && manager()->isSelectable(obj)) {
        manager->setSelected(!manager->isSelected(obj));
    }
}
```

Button-Up Events

If the event comes from the left button, `handleButtonUp` is called. Otherwise, the event is dispatched to accelerators.

```
case IlvButtonUp:
    if (event.button() == IlvLeftButton)
        handleButtonUp(IlvPoint(event.x(), event.y()));
    else getManager()->shortCut(event, getView());
    break;
```

Button-Dragged Events

The `handleButtonDragged` member function is called, but only if the event comes from the left button.

```
case IlvButtonDragged:
    if (event.modifiers() == IlvLeftButton){
        IlvPoint p(event.x(), event.y());
        handleButtonDragged(p);
    }
    break;
```

drawGhost Member Function

This member function is split in three parts: the common part, which is the entry point from the member function `handleEvent`, and two others, depending on the type of translation being done.

If there is only one selected object, a specific `drawGhost` is called for this object. Otherwise, another `drawGhost` function that handles a rectangle is called:

```
void
IlvMoveInteractor::drawGhost()
{
    if (!_xor_rectangle.w()) return;
    if (manager()->numberOfSelections() == 1)
        drawGhost(_move);
    else
        drawGhost(_xor_rectangle);
```

drawGhost for a Rectangle

This member function is called if there is more than one selected object. It displays the bounding box of all the selected objects being moved in the view. The palette of the IlvManager object is used:

```
void
IlvMoveInteractor::drawGhost(const IlvRect& rect, IlvRegion* clip)
{
    if (!rect.w()) return;
    IlvManager* manager = getManager();
    if (clip) manager->getPalette()->setClip(clip);

    getView()->drawRectangle(manager->getPalette(),rect);
    if (clip) manager->getPalette()->setClip();
}
```

drawGhost for an Object

This member function is called if there is only one selected object. It displays the object at its new coordinates by calling the draw member function after its palette has been set to XOR mode. The new coordinates are computed from the difference between the coordinates of the rectangle being dragged and the coordinates of the original bounding box of the object:

```
void
IlvMoveInteractor::drawGhost(IlvGraphic* obj, IlvRegion* clip)
{
    if (!getManager()->isMoveable(obj) || !_xor_rectangle.w())
        return;
    IlvPos tempdx, tempdy;
    if (getTransformer()) {
        IlvRect r1(_xor_rectangle);
        IlvRect r2(_bbox);
        getTransformer()->inverse(r1);
        getTransformer()->inverse(r2);
        tempdx = r1.x() - r2.x();
        tempdy = r1.y() - r2.y();
    } else {
        tempdx = _xor_rectangle.x() - _bbox.x();
        tempdy = _xor_rectangle.y() - _bbox.y();
    }
    obj->translate(tempdx, tempdy);
    obj->setMode(IlvModeXor);
    obj->draw(getView(), getTransformer(), clip);
    obj->setMode(IlvModeSet);
    obj->translate(-tempdx, -tempdy);
}
```

doIt Member Function

The doIt member function must apply the translation to all selected objects. The delta parameter gives the translation vector expressed in the view coordinate system so it must be converted to the object coordinate system. Then the objects must be translated. This cannot be done by calling the IlvGraphic member functions directly; it must be done by the

manager. Here, `IlvManager::applyToSelections` calls `TranslateObject` for each selected object:

```
void
TranslateObject(IlvGraphic* object, IlvAny argDelta)
{
    IlvPoint* delta = (IlvPoint*)argDelta;
    object->translate(delta.x(), delta.y());
}

void
IlvMoveInteractor::doIt(const IlvPoint& delta)
{
    IlvPoint origin(0, 0),
            tdelta(delta);
    if (getTransformer()) {
        getTransformer()->inverse(origin);
        getTransformer()->inverse(tdelta);
    }
    IlvPoint dp(tdelta.x()-origin.x(),
                tdelta.y()-origin.y());
    getManager->applyToSelections(TranslateObject, &dp);
}
```

handleButtonDown Member Function

The `handleButtonDown` member function selects the object to be moved, storing its previous state in `_wasSelected`. Then, it computes the `_bbox` field by means of a call to the `ComputeBBoxSelections` function. This function returns in `_bbox` the bounding box of all the selected objects:

```
static void
ComputeBBoxSelections(IlvManager* manager, IlvRect& bbox, IlvView* view)
{
    bbox.resize(0, 0);
    IlUInt nbselections;
    IlvGraphic** objs = manager->getSelections(nbselections);
    IlvRect rect;
    IlvTransformer* t = manager->getTransformer(view);
    for (IlUInt i=0; i < nbselections; i++) {
        objs[i]->boundingBox(rect, t);
        bbox.add(rect);
    }
}
void
IlvMoveInteractor::handleButtonDown(const IlvPoint& p)
{
    IlvGraphic* obj = getManager()->lastContains(p, getView());
    if (!obj) return;
    IlvDrawSelection* sel = manager()->getSelection(obj);
    if (!sel && getManager()->isSelectable(obj)) {
        getManager()->deSelect();
        getManager()->makeSelected(obj);
        _wasSelected = IlFalse;
        sel = getManager()->getSelection(obj);
    } else
        _wasSelected = IlTrue;
```

```
    if (sel) {
        ComputeBBoxSelections(getManager(), _bbox, getView());
        _move   = obj;
        _deltax = _bbox.x() - p.x();
        _deltay = _bbox.y() - p.y();
    }
}
```

The `ComputeBBoxSelections` section is described in more detail.

The first part initializes the result to the empty rectangle, and then queries the manager for all the selected objects. `nbselections` is the number of selected objects in the array `objs`:

```
bbox.resize(0, 0);
IlUInt nbselections;
IlvGraphic** objs = manager->getSelections(nbselections);
```

The next part starts a loop to scan every object:

```
IlvRect rect;
for (IlUInt i=0; i < nbselections; i++) {
```

This next part reads the bounding box of each object, transformed in the view coordinate system, and adds it to the result:

```
objs[i]->boundingBox(rect, t);
for (IlUInt i=0; i < nbselections; i++) {
    objs[i]->boundingBox(rect, t);
    bbox.add(rect);
}
```

handleButtonDragged Member Function

If there is a moving object and if it is moveable, the dragging position is snapped to the manager grid (if one exists) and a new `_xor_rectangle` is computed. Then, the member function `ensureVisible` makes sure that the point the user drags will remain on the visible part of the view:

```
void
IlvMoveInteractor::handleButtonDragged(const IlvPoint& point)
{
    if (!_move) return;
    IlvPoint p = point;
    IlvRect  rect;
    if (getManager()->isMoveable(_move)) {
        if (_xor_rectangle.w()) drawGhost();
        p.translate(_deltax, _deltay);
        getManager()->snapToGrid(getView(), p);
        p.translate(-_deltax, -_deltay);
        _xor_rectangle.move(p.x() + _deltax, p.y() + _deltay);
        _xor_rectangle.resize(_bbox.w(), _bbox.h());
        ensureVisible(p);
        drawGhost();
    }
}
```

handleButtonUp Member Function

If there are objects to move, they are translated by calling the member function `doIt`. Otherwise, the last designated object is deselected:

```
void
IlvMoveInteractor::handleButtonUp(const IlvPoint&)
{
    if (!_move) return;
    IlvDrawSelection* sel = getManager()->getSelection(_move);
    if (_move && _xor_rectangle.w() && sel) {
        drawGhost();
        IlvDeltaPoint delta(_xor_rectangle.x() - _bbox.x(),
                            _xor_rectangle.y() - _bbox.y());
        _xor_rectangle.w(0);
        _move = 0;
        doIt(delta);
    } else {
        _xor_rectangle.w(0);
        _move = 0;
        if (sel && _wasSelected) getManager()->deSelect();
    }
}
```

Object Interactors

The `IlvManagerObjectInteractor` class is deprecated since IBM ILOG Views 4.0.

For a description of how to use object interactors, see section *Managing Events: Object Interactors* in *Chapter 8, IlvContainer: The Graphic Placeholder Class* of the IBM ILOG Views *Foundation User's Manual*.

Accelerators

An accelerator is a simple binding of an event description with an application function called the *accelerator action*. Accelerators provide a quick way of attaching a behavior to a manager, but the interaction is basic; it only involves one event (for instance, a key press or a mouse click).

An accelerator is not bound to a particular view or graphic object; it can be triggered in any view or any object of the manager. However, accelerators come last in the manager event dispatching mechanism. They can only be activated if event hooks, view interactors, and object interactors have not intercepted the event.

The accelerator action must be defined as an `IlvManagerAcceleratorAction`:

```
typedef void (* IlvManagerAcceleratorAction)(IlvManager*, IlvView*,
                                             IlvEvent&, IlvAny);
```

The following `IlvManager` member functions allow you to manipulate manager accelerators:

- `IlvManager::addAccelerator`
- `IlvManager::getAccelerator`
- `IlvManager::removeAccelerator`
- `IlvManager::shortCut`

The `IlvManager::shortCut` member function is called to dispatch an event to accelerators. If an accelerator event description matches the event to dispatch, the accelerator action is called.

Example: Changing the Key Assigned to an Accelerator

The code below assigns the Ctrl-F key instead of ‘f’ to the action `IlvManager::fitTransformerToContents`.

```
IlvManagerAcceleratorAction action;
IlvAny arg;
if (manager->getAccelerator(&action, &arg, IlvKeyUp, ‘f’))
{
    manager->addAccelerator(action,
                            IlvKeyUp,
                            IlvCtrlChar(‘f’),
                            0,
                            arg);
    manager->removeAccelerator(IlvKeyUp, ‘f’);
}
```

Predefined Manager Accelerators

Managers have built-in accelerators, which are listed below. You can disconnect them by setting the `accelerators` parameter of the manager constructor to `IlFalse`.

Table 2.1 *Predefined Manager Accelerators*

Event Type	Key or Button	Action
`IlvKeyUp`	f	Modifies the zoom factor of the view so that all objects can be seen (f for fit).
`IlvKeyUp`	i	Sets the transformer of this view to the identity matrix.
`IlvKeyUp`	p	Moves selected objects to a higher layer.
`IlvKeyUp`	P	Moves selected objects to a lower layer.

Table 2.1 Predefined Manager Accelerators (Continued)

Event Type	Key or Button	Action
IlvKeyUp	Ctrl-D	Duplicates all selected objects and moves the copied objects slightly.
IlvKeyUp	Ctrl-A	Selects all objects.
IlvKeyUp	Ctrl-S	Selects the object designated by the pointing device.
IlvKeyUp	Del	Deletes all selected objects.
IlvKeyDown	r	Re-executes the last command.
IlvKeyDown	u	Undoes the last command.
IlvKeyUp	Ctrl-G	Groups the selected objects into an IlvGraphicSet.
IlvKeyUp	Ctrl-U	Ungroups an IlvGraphicSet.
IlvKeyDown	Right	Translates the view left.
IlvKeyDown	Left	Translates the view right.
IlvKeyDown	Down	Translates the view up.
IlvKeyDown	Up	Translates the view down.
IlvKeyUp	Z	Zooms into the view.
IlvKeyUp	U	Zooms out of the view.
IlvKeyUp	Ctrl-B	Deselects all objects.
IlvKeyUp	Ctrl-T	Inverts all selected objects.
IlvKeyUp	Y	Flips the selected objects horizontally.
IlvKeyUp	y	Flips the selected objects vertically.
IlvKeyUp	. (dot)	Flips the selected objects both horizontally and vertically.
IlvKeyUp	Ctrl-C	Copies selected objects on the clipboard.
IlvKeyDown	Ctrl-V	Inserts objects from the clipboard.
IlvKeyUp	Ctrl-X	Deletes selected objects but saves them on the clipboard.

Table 2.1 Predefined Manager Accelerators (Continued)

Event Type	Key or Button	Action
IlvKeyDown	R	Rotates the view 90 degrees counter-clockwise.
IlvKeyDown	C	Centers the view on the indicated point.
IlvKeyUp	T	Encapsulates relevant object in IlvTransformer graphic(s).

By means of calls to `IlvManager::getAccelerator`, you can reassign these keys to fit your own application needs. You can also add your own interactors to this primary list, remove any of them, or overload them so they act differently.

CHAPTER

3

Advanced Manager Features

This section describes more advanced features of managers. These are as follows:

- *Observers*
- *View Hooks*
- *Manager Grid*
- *Undoing and Redoing Actions*

Observers

Applications can be notified when the state of a manager changes. This notification mechanism is based on `IlvManagerObserver`, a subclass of `IlvObserver`. Observers are created by the application and set on the manager. The manager is in charge of sending messages to the observer under certain circumstances called *reasons*.

Notification messages are classified by their reason into different categories. An observer can choose to receive messages of one or several categories by setting its *interest mask*. The

manager will only send a message to the observer if the notification reason belongs to a category of the observer interest mask. These categories are shown in Table 3.1:

Table 3.1 *Notification Categories*

Category Description	Mask
General	IlvMgrMsgGeneralMask
Manager view	IlvMgrMsgViewMask
Manager layer	IlvMgrMsgLayerMask
Manager contents	IlvMgrMsgContentsMask
Object geometry	IlvMgrMsgObjectGeometryMask

An application wishing to get notification messages must define a subclass of `IlvManagerObserver` and overload the virtual member function `update`. In this member function, the observer receives an instance of `IlvManagerMessage`, or a subclass, containing the reason and additional relevant information about the notification.

General Notifications

This category concerns general notifications on the managers.

Interest mask: `IlvMgrMsgGeneralMask`

- Delete the manager

  Reason: `IlvMgrMsgDelete`

  Message type: `IlvManagerMessage`

Manager View Notifications

This category concerns notifications on operations performed on manager views.

Interest mask: `IlvMgrMsgViewMask`

- Add a view to the manager

  Reason: `IlvMgrMsgAddView`

  Message type: `IlvManagerAddViewMessage`

- Remove a view from the manager

  Reason: `IlvMgrMsgRemoveView`

  Message type: `IlvManagerRemoveViewMessage`

- Set an interactor on a view

Reason: `IlvMgrMsgSetInteractor`

Message type: `IlvManagerSetInteractorMessage`

- Set a transformer on a view

  Reason: `IlvMgrMsgSetTransformer`

  Message type: `IlvManagerSetTransformerMessage`

Manager Layer Notifications

This category concerns notifications on operations performed on manager layers.

Interest mask: `IlvMgrMsgLayerMask`

- Add a layer to the manager

  Reason: `IlvMgrMsgAddLayer`

  Message type: `IlvManagerLayerMessage`

- Remove a layer from the manager

  Reason: `IlvMgrMsgRemoveLayer`

  Message type: `IlvManagerLayerMessage`

- Change the index of a layer

  Reason: `IlvMgrMsgMoveLayer`

  Message type: `IlvManagerMoveLayerMessage`

- Swap indexes between two layers

  Reason: `IlvMgrMsgSwapLayer`

  Message type: `IlvManagerSwapLayerMessage`

- Set the name of a layer

  Reason: `IlvMgrMsgLayerName`

  Message type: `IlvManagerLayerNameMessage`

- Set the visibility of a layer

  Reason: `IlvMgrMsgLayerVisibility`

  Message type: `IlvManagerLayerVisibilityMessage`

- Set the selectabililty of a layer

  Reason: `IlvMgrMsgLayerSelectability`

  Message type: `IlvManagerLayerMessage`

Manager Contents Notifications

This category concerns notifications on the changes in the contents of managers.

Interest mask: `IlvMgrMsgContentsMask`

- Add a graphic object to the manager

  Reason: `IlvMgrMsgAddObject`

  Message type: `IlvManagerContentsMessage`

- Remove a graphic object from the manager

  Reason: `IlvMgrMsgRemoveObject`

  Message type: `IlvManagerContentsMessage`

- Set the layer of a graphic object

  Reason: `IlvMgrMsgObjectLayer`

  Message type: `IlvManagerObjectLayerMessage`

Graphic Object Geometry Notifications

This category concerns notifications on a change of geometry of the objects (for example, move, resize, and rotate).

Interest mask: `IlvMgrMsgObjectGeometryMask`

- Change the geometry of a graphic object

  Reason: `IlvMgrMsgObjectGeometry`

  Message type: `IlvManagerObjectGeometryMessage`

Example

Here is the implementation of an observer that receives notifications on adding or removing layers and views.

```
class MyManagerObserver
: public IlvManagerObserver
{
public:
    MyManagerObserver(IlvManager* manager)
    : IlvManagerObserver(manager,
                         IlvMgrMsgLayerMask | IlvMgrMsgViewMask)
    {}
    virtual void update(IlvObservable* o, IlvAny arg);
};
```

The `update` member function:

```
void MyManagerObserver::update(IlvObservable* obs, IlvAny arg)
{
    IlvManager* manager = ((IlvManagerObservable*)obs)->getManager();
    switch(((IlvManagerMessage*) arg)->_reason) {
      // __ Notification on manager view
      case IlvMgrMsgAddView:
          IlvPrint("Add view notification");
          break;
      case IlvMgrMsgRemoveView:
          IlvPrint("Remove view notification");
          break;
      // __ Notification on manager layer
      case IlvMgrMsgAddLayer:
          IlvPrint("Add layer notification: %d",
                   ((IlvManagerLayerMessage*)arg)->getLayer());
          break;
      case IlvMgrMsgRemoveLayer:
          IlvPrint("Remove layer notification: %d",
                   ((IlvManagerLayerMessage*)arg)->getLayer());
          break;
      default:
          IlvPrint("Unhandled notification");
          break;
    }
}
```

To attach the observer to the manager:

```
MyManagerObserver* observer = new MyManagerObserver(manager);
```

View Hooks

Manager view hooks are part of a mechanism allowing the application to be notified when certain actions are performed on or by the manager. This can be used for various reasons such as monitoring the contents of a manager, performing additional drawings when the manager redraws its graphic objects, or taking an action when the transformer of a manager view changes.

Note: *Another notification mechanism is described in section Observers.*

This section is divided as follows:

- *Manager View Hooks*
- *Example: Monitoring the Number of Objects in a Manager*
- *Example: Maintaining a Scale Displayed With No Transformation*

Manager View Hooks

A manager view hook is an instance of the `IlvManagerViewHook` class. To be active, it must be associated with a manager view. Each manager view handles a list of view hooks. To connect and disconnect view hooks from a manager view, use the following `IlvManager` member functions:

- `IlvManager::installViewHook`
- `IlvManager::removeViewHook`

The `IlvManagerViewHook` class has a number of virtual member functions that are automatically called by the manager when certain predefined operations occur. Here is the list of these member functions and the circumstances under which they are called:

- `IlvManagerViewHook::beforeDraw`

  Called before the manager draws in the manager view. Applications often overload this member function to perform additional drawings before the manager displays its graphic objects.

- `IlvManagerViewHook::afterDraw`

  Called after the manager has drawn in the manager view. Applications often overload this member function to perform additional drawings on top of the graphic objects displayed by the manager.

- `IlvManagerViewHook::afterExpose`

  Called after the manager has received an Expose event in the view.

- `IlvManagerViewHook::interactorChanged`

  Called when the interactor of the manager view changes.

- `IlvManagerViewHook::transformerChanged`

  Called when the transformer of the manager view changes.

- `IlvManagerViewHook::viewResized`

  Called when the manager view is resized.

- `IlvManagerViewHook::viewRemoved`

  Called when the manager view is detached from the manager.

- `IlvManagerViewHook::contentsChanged`

  Called when the contents of the manager change, that is, graphic objects have been added, removed, or their geometry has changed.

When an event occurs in view, the manager calls the corresponding member functions of all the hooks attached to this view.

Example: Monitoring the Number of Objects in a Manager

The following code is a subclass of `IlvManagerViewHook` that displays in an `IlvTextField` the number of objects contained in the manager:

```
class DisplayObjectsHook
: public IlvManagerViewHook
{
public:
    DisplayObjectsHook(IlvManager* manager,
                       IlvView* view,
                       IlvTextField* textfield)
    : IlvManagerViewHook(manager, view),
      _textfield(textfield)
      {}
    virtual void contentsChanged();
protected:
    IlvTextField* _textfield;
};


void DisplayObjectsHook::contentsChanged()
{
    IlvUInt count = getManager()->getCardinal();
    _textfield->setValue((IlvInt)count, IlvTrue);
}
```

Example: Maintaining a Scale Displayed With No Transformation

This part presents an example of subtyping an `IlvManagerViewHook`. At first there is a map and a circular scale used as a compass card. Then, because of hooks, the manager translates and zooms the view without affecting the compass card. The `IlvManagerViewHook::afterDraw` and `IlvManagerViewHook::transformerChanged` member functions are redefined to redraw the scale to its original dimensions and location.

```
static void ILVCALLBACK
Quit(IlvView* view, IlvAny)
{
    delete view->getDisplay();
    IlvExit(0);
}

char* labels[] = {"N", "O", "S", "E", ""};

class ExHook
: public IlvManagerViewHook
{
public :
    ExHook(IlvManager* m, IlvView* v, const IlvRect* psize=0)
    : IlvManagerViewHook(m, v)
    {
        _cirscale = new IlvCircularScale(m->getDisplay(),
```

```
                                         IlvRect(30, 30, 100, 100),
                                         "%.4f",
                                         0, 100, 90., 360.);
        _cirscale->setLabels(5, (const char* const*)labels);
    }
    virtual void afterDraw(IlvPort*,
                           const IlvTransformer* = 0,
                           const IlvRegion* = 0,
                           const IlvRegion* = 0);
    virtual void transformerChanged(const IlvTransformer*,
                                    const IlvTransformer*);
protected :
    IlvRect _size;
    IlvCircularScale* _cirscale;
};
void ExHook::afterDraw(IlvPort* dst,
                       const IlvTransformer*,
                       const IlvRegion*,
                       const IlvRegion* clip)
{
    if (getManager()->isInvalidating())
        getManager()->reDrawViews();
    _cirscale->draw(dst, 0, 0 /*clip*/);
    if (dst->isABitmap())
        _cirscale->draw(getView(), 0, 0);
}
void ExHook::transformerChanged(const IlvTransformer* current,
                                const IlvTransformer* old)
{
    IlvRect bbox;
    _cirscale->boundingBox(bbox);
    if (old) old->inverse(bbox);
    if (current) current->apply(bbox);
    if (!getManager()->isInvalidating())
    {
        getManager()->initReDraws();
        getManager()->invalidateRegion(getView(), bbox);
    }
}

static void
SetDoubleBuffering(IlvManager* m,
                   IlvView* v,
                   IlvEvent&,
                   IlvAny)
{
    m->setDoubleBuffering(v, !m->isDoubleBuffering(v));
}

int
main(int argc, char* argv[])
{
    IlvDisplay* display = new IlvDisplay("Example", "", argc, argv);
    if (!display || display->isBad())
    {
       IlvFatalError("Can't open display");
       IlvExit(-1);
    }
```

```
    IlvView* view = new IlvView(display, "ExMan", "Manager",
                                IlvRect(0, 0, 400, 400));
    view->setDestroyCallback(Quit);
    IlvManager* manager = new IlvManager(display);
    manager->addView(view);
    manager->addAccelerator(SetDoubleBuffering, IlvKeyUp, 'b');

    // Description of a map
    manager->read("../hook.ilv");

    // Instantiation of the hook class
    ExHook* pHook = new ExHook(manager, view);

    // Connect the hook to the manager view
    manager->installViewHook(pHook);
    manager->setInteractor(new IlvSelectInteractor(manager, view));

    IlvMainLoop();
}
```

Manager Grid

Most editors provide a snapping grid that forces mouse events to occur at specified locations. Usually, the coordinates where the user can move the pointing device are located at grid points. If the manager is configured to allow standard mouse events, all event locations can be automatically modified so they occur only at specific locations. Thus, the effect of filtering user events by a manager grid is to modify their locations to the closest grid point.

The `IlvManagerGrid` class is responsible for the conversion to a valid grid point of the coordinates of an event that occurs in a view.

You can set or remove a snapping grid in each of the views handled by a manager. You can configure these grids to make them:

- visible or not visible,
- active or inactive.

You can also make the grid take on different shapes by subtyping the `IlvManagerGrid` class. The default implementation is a rectangular grid for which you can set the origin and the horizontal and vertical spacing values.

When a grid is made visible, it draws dots with the color specified as the foreground color of the `palette` parameter.

The grid can be made invisible when it is created by setting the `visible` parameter to `IlFalse`. To make the grid initially inactive, set the `active` parameter to `IlFalse`.

To display only a subset of the grid points, use the last two `IlvDim`-typed parameters. These indicate the nature of the subset, that is, one out of every quantity of dots along the horizontal and vertical axes is displayed in the given direction. However, the event location snapping takes place on each of the grid points, whether shown or not.

Example: Using a Grid

This code sets a new grid to the view `view` associated with the manager:

```
// Get the previous grid
IlvManagerGrid* previousGrid = manager->getGrid(view);

// Create a new instance of IlvManagerGrid
IlvManagerGrid* newGrid = new IlvManagerGrid(display->getPalette(),
                                             IlvPoint(0, 0),
                                             10,
                                             10);

// Set the new grid to the view
manager->setGrid(view, newGrid);

// If a previous grid existed then delete it
if (previousGrid)
    delete previousGrid;
```

Usually, it is not necessary to delete a previous grid, since by default none is associated with the view.

The following code shows how to create an `IlvLine` whose ends are on the grid:

```
static void
AddSnappedLine(IlvManager* manager,
               const IlvView* view,
               const IlvPoint& start,
               const IlvPoint& end)
{
    IlvPoint p1 = start;
    IlvPoint p2 = end;

    // Compute the new coordinates
    manager->snapToGrid(view, p1);
    manager->snapToGrid(view, p2);

    // Create an object IlvLine
    IlvGraphic* object = new IlvLine(manager->getDisplay(), p1, p2);

    // Add the object to the manager
    manager->addObject(object);
}
```

All the standard interactors of IBM ILOG Views that create graphic objects use `IlvManager::snapToGrid`.

Undoing and Redoing Actions

This section describes how to implement the undo/redo process with the `IlvManagerCommand` class.

In order to remember every action that your program user may apply to objects (and the objects as well), the manager creates specific instances of the `IlvManagerCommand` class, depending on what kind of action was required. The manager can then manipulate a stack of these commands. A request for `IlvManager::unDo` pops an item off the stack, and applies the inverse operation that created the popped item.

The `IlvManager::reDo` operation duplicates the topmost item of the command stack and executes the operation again.

This section is divided as follows:

- *Command Class*
- *Managing Undo*
- *Example: Using the IlvManagerCommand Class to Undo/Redo*
- *Managing Modifications*

Command Class

Each ready-to-use command in IBM ILOG Views was implemented with the `IlvManagerCommand` class. To carry out undo/redo operations, the subtypes of this class merely store the arguments of commands. The actual command to be remembered is known by the type of the `IlvManagerCommand` objects.

If you create a new operation for the manager and you want to undo and redo it, you have to create a specific subtype of the `IlvManagerCommand` class. A complete example of this subtyping is described in *Example: Using the IlvManagerCommand Class to Undo/Redo*.

> ***Note:*** *All predefined interactors use the* `IlvManagerCommand` *class. Therefore, it is possible to undo and redo their effect.*

Managing Undo

The following `IlvManager` member functions handle undo operations:

- `IlvManager::addCommand`
- `IlvManager::isUndoEnabled`
- `IlvManager::setUndoEnabled`
- `IlvManager::forgetUndo`

- `IlvManager::reDo`
- `IlvManager::unDo`

Each action applied to manager objects is inserted in a special queue maintained by each `IlvManager` instance. The undo/redo process is based on this queue management.

Example: Using the IlvManagerCommand Class to Undo/Redo

This subsection shows the implementation of the `IlvTranslateObjectCommand` class, subclass of `IlvManagerCommand`:

The constructor of this class stores the parameters of the translation operation:

```
IlvTranslateObjectCommand::IlvTranslateObjectCommand(IlvManager*     manager,
                                                     IlvGraphic*     object,
                                                     const IlvPoint& dp)
: IlvManagerCommand(manager),
  _dx(dp.x()),
  _dy(dp.y()),
  _object(object)
{}
```

doIt Member Function

The `IlvTranslateObjectCommand::doIt` member function is implemented as follows:

```
void
IlvTranslateObjectCommand::doIt()
{
    _manager->translateObject(_object, _dx, _dy, IlvTrue);
}
```

The operation to be performed is the translation of the object by `_dx` and `_dy`.

unDo Member Function

The `IlvTranslateObjectCommand::unDo` member function is as follows:

```
void
IlvTranslateObjectCommand::unDo()
{
    _manager->translateObject(_object, -_dx, -_dy, IlvTrue);
}
```

The inverse translation is applied and the regions are redrawn.

copy Member Function

The `IlvTranslateObjectCommand::copy` member function creates a copy of the command object and returns it.

```
IlvManagerCommand*
IlvTranslateObjectCommand::copy() const
{
    return new IlvTranslateObjectCommand(_manager, _object, _dx, _dy);
}
```

Managing Modifications

The following `IlvManager` member functions let you manage the state of objects (modified or not) handled by the manager:

- `IlvManager::isModified`
- `IlvManager::setModified`
- `IlvManager::contentsChanged`

Example: Setting the State of a Manager to Unmodified

```
manager->setModified(IlFalse);
```

There are also two global functions:

- `IlvGetContentsChangedUpdate`
- `IlvSetContentsChangedUpdate`

Example: Disallowing View Hook Calls in contentsChanged

The following code disallows the calls to the `IlvManager::contentsChanged` member functions of the existing view hooks associated with the manager view:

```
IlvSetContentsChangedUpdate(IlTrue);
```

Part II

Grapher

Part II describes a high-level IBM® ILOG® Views package called the grapher, which is dedicated to the graphic representation of hierarchical and interconnected information. This part consists of the following chapters:

- **Chapter 1, *Introducing the Grapher Extension of IBM ILOG Views Studio*** describes how to use the Grapher extension of IBM ILOG Views Studio.
- **Chapter 2, *Features of the Grapher Package*** describes the classes, methods, and principles that make the Grapher package work.

> ***Note:*** *The IBM ILOG Views Grapher package is available only if you have purchased the IBM ILOG Views 2D Graphics Professional product.*

C H A P T E R

1

Introducing the Grapher Extension of IBM ILOG Views Studio

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

- ◆ The Main Window
- ◆ The Palettes Panel
- ◆ Grapher Extension Commands

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

The Main Window

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

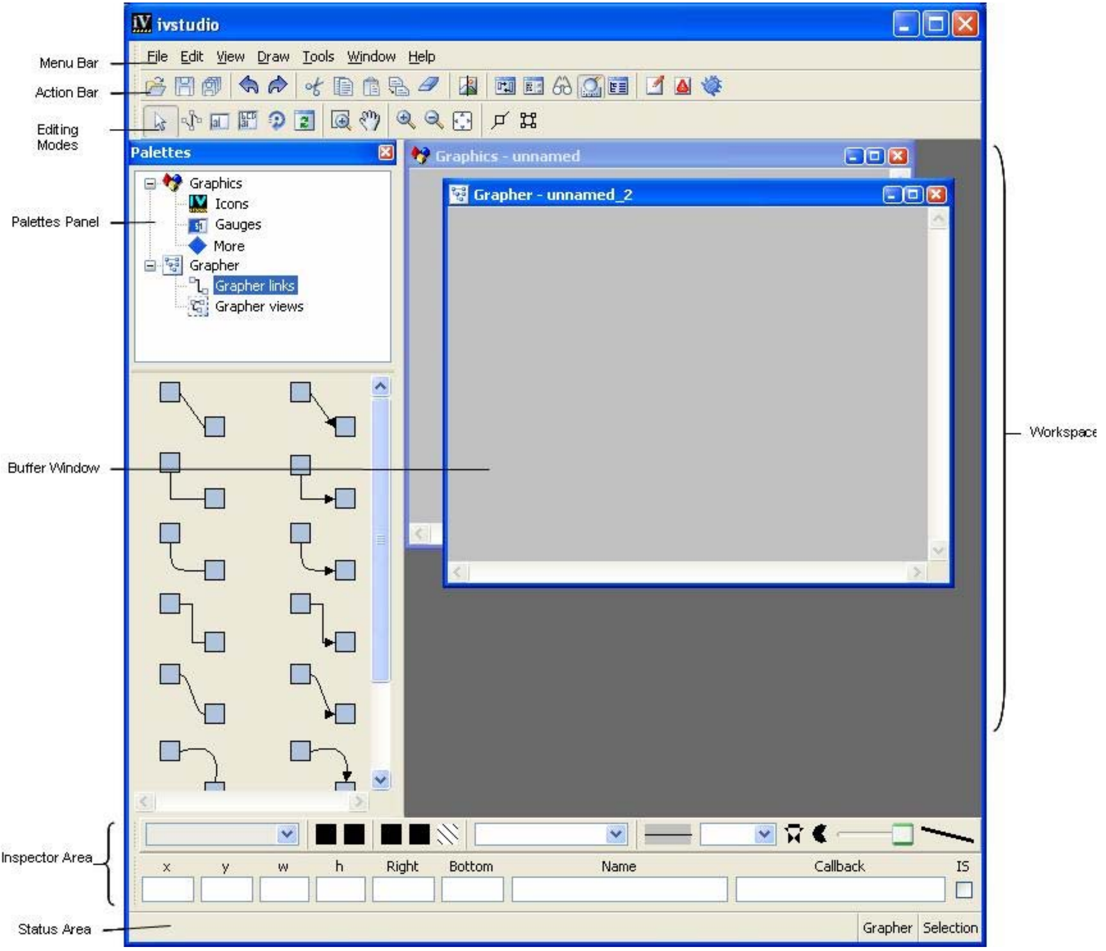


Figure 1.1 IBM ILOG Views *Studio Main Window with Grapher Extension at Start-up*

The Main window appears much as it does when only the Foundations package is installed. However, you will notice that with the Grapher package you have access to an additional buffer window, 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 Grapher extension of IBM® ILOG® Views Studio, you can edit the following types of buffers:

- Grapher
- 2D Graphics

An empty Graphics buffer is displayed by default when you launch IBM ILOG Views Studio.

> ***Note:*** *You will notice the following difference as you switch between the different types of buffers 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 Grapher Buffer Window

The Grapher buffer window lets you display and edit graphs. It uses an `IlvGrapher` to load, edit, and save nodes and links.

To create a new Grapher buffer window:

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

To open this window, you can also execute the `NewGrapherBuffer` 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 `IlvGrapher`, a Grapher buffer window is automatically opened.

The 2D Graphics Buffer Window

The 2D Graphics buffer is the default for the Foundation package. It is still available with the Grapher 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 Menu Bar

When the Grapher package is installed, an additional command is available through the menu bar in the Main window:

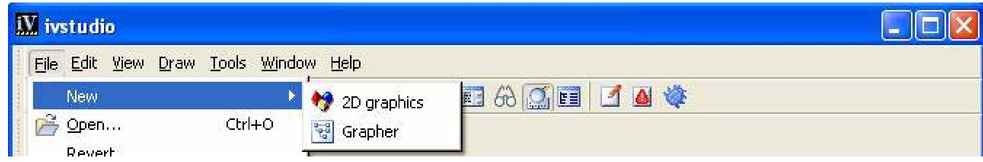


Figure 1.2 IBM ILOG Views Studio Grapher Extension Menu Bar

In the menu File > New, there is now the menu item Grapher, which creates a new Grapher buffer. This is the command NewGrapherBuffer.

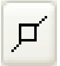
The Action Toolbar

The Action toolbar remains unchanged from the Foundation package:

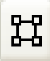
The Editing Modes Toolbar

The Editing Modes toolbar appears as follows when the Grapher buffer is the active window in the work space:

Figure 1.3 IBM ILOG Views Studio Grapher Extension Editing Modes Toolbar



Make Node - Use this button to make the selected objects into nodes. It implements the `MakeNode` command.



Pin Editor Mode - Use this mode to interactively edit the connection pins defined on grapher nodes. For more information on how you can use this mode, please refer to Editing Connection Pins.

The Palettes Panel

When using the Grapher extension of IBM® ILOG® Views Studio, you have access to the Grapher links through the Palettes panel.

You will notice in the upper pane of the Palettes panel two additional palettes that are provided with the Grapher extension. Click the appropriate palette in the upper pane to display the various Grapher links in the lower pane:

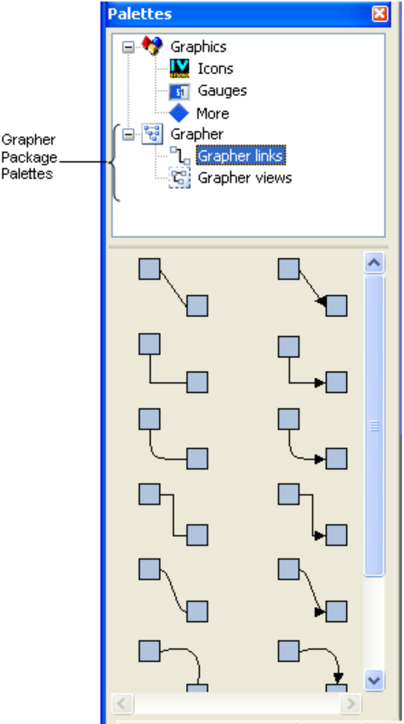


Figure 1.4 *IBM ILOG Views Studio Grapher Extension Palettes Panel*

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

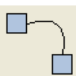
The Grapher Palettes

The Grapher palettes contain the following objects that can be used to create Grapher links. (Links can also be created by using link edit commands from the command panel.)
To select a linking mode, click on the link itself between the two `IlvShadowRectangles`, and the link will appear bounded with an orange box.

These modes can only be used in a Grapher buffer.

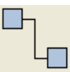
> ***Note:*** *A Grapher link can only be created between nodes, therefore the objects to be linked must first be declared as nodes using the* `MakeNode` *command. First select the objects and then click the Make Node button on the Editing Modes toolbar.*

ArcLinkImage



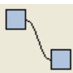
Use this mode to link two grapher nodes with an `IlvArcLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

DoubleLinkImage



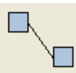
Use this mode to link two grapher nodes with an `IlvDoubleLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

DoubleSplineLinkImage



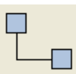
Use this mode to link two grapher nodes with an `IlvDoubleSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

LinkImage



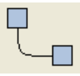
Use this mode to link two grapher nodes with an `IlvLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OneLinkImage



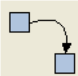
Use this mode to link two grapher nodes with an `IlvOneLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OneSplineLinkImage



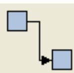
Use this mode to link two grapher nodes with an `IlvOneSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedArcLinkImage



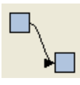
Use this mode to link two grapher nodes with an oriented `IlvArcLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedDoubleLinkImage



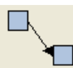
Use this mode to link grapher nodes with an oriented `IlvDoubleLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedDoubleSplineLinkImage



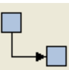
Use this mode to link selected grapher nodes with an oriented `IlvDoubleSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedLinkImage



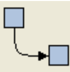
Use this mode to link two grapher nodes with an oriented `IlvLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedOneLinkImage



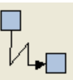
Use this mode to link two grapher nodes with an oriented `IlvOneLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedOneSplineLinkImage



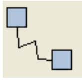
Use this mode to link grapher nodes with an oriented `IlvOneSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedPolylineLinkImage



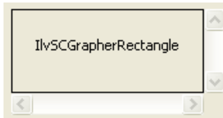
Use this mode to link grapher nodes with an oriented `IlvPolylineLinkImage` object. Click on the first node, then on intermediate points as required, and double-click on the second node to finish the operation.

PolylineLinkImage



Use this mode to link grapher nodes with an `IlvPolylineLinkImage` object. Click on the first node, then on intermediate points as required, and double-click on the second node to finish the operation.

IlvSCGrapherRectangle



This creates an `IlvSCGrapherRectangle` object to display the contents of an `IlvGrapher`. Use either the drag-and-drop operation or the creation mode operation. (This command is found in the Grapher Views palette.)

Grapher Extension Commands

This section presents an alphabetical listing of the additional, predefined commands that are available in the Grapher extension of IBM® ILOG® Views Studio. (All of the IBM ILOG Views Studio Foundation commands are also available.) For each command, it indicates its label, how to access it if it is accessible other than through the Commands panel, the category to which it belongs, 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.

MakeNode

Label	Node
Path	Main window: Editing Modes toolbar when editing Grapher buffers.
Category	grapher, studio
Action	If the current buffer is a Grapher buffer, this command makes the selected objects into nodes.

NewGrapherBuffer

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

SelectArcLinkImageMode

Label	Arc-shaped link
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates an arc-shaped link between two nodes. See section *IlvArcLinkImage*.

SelectDoubleLinkImageMode

Label	DoubleLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a two-bend link between two nodes. See section *IlvDoubleLinkImage*.

SelectDoubleSplineLinkImageMode

Label	DoubleSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a two-bend curved link between two nodes. See section *IlvDoubleSplineLinkImage*.

SelectLinkImageMode

Label	LinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a direct link between two nodes. See section *Base Class for Links*.

SelectOneLinkImageMode

Label	OneLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a one-bend link between two nodes. See section *IlvOneLinkImage*.

SelectOneSplineLinkImageMode

Label	OneSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a one-bend curved link between two nodes. See section *IlvOneSplineLinkImage*.

SelectOrientedArcLinkImageMode

Label	Oriented Arc-shaped link
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented arc-shaped link between two nodes. See section *IlvArcLinkImage*.

SelectOrientedDoubleLinkImageMode

Label	Oriented DoubleLinkImage
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates an oriented two-bend link between two nodes. See section *IlvDoubleLinkImage*.

SelectOrientedDoubleSplineLinkImageMode

Label	Oriented DoubleSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented two-bend curved link between two nodes. See section *IlvDoubleSplineLinkImage*.

SelectOrientedLinkImageMode

Label	Oriented LinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented direct link between two nodes. See section *Base Class for Links*.

SelectOrientedOneLinkImageMode

Label	Oriented OneLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented one-bend link between two nodes. See section *IlvOneLinkImage*.

SelectOrientedOneSplineLinkImageMode

Label	Oriented OneSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented one-bend curved link between two nodes. See section *IlvOneSplineLinkImage*.

SelectOrientedPolylineLinkImageMode

Label	Free-shape oriented link
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented free-shaped link between two nodes. See section *IlvPolylineLinkImage*.

SelectPinEditorMode

Label	PinEditor
Path	Main window: Editing Modes toolbar when editing Grapher buffers.
Category	grapher
Action	Sets the Pin editing mode on the current buffer. See section Editing Connection Pins.

SelectPolylineLinkImageMode

Label	Free-shape link
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates a free-shaped link between two nodes. See section *IlvPolylineLinkImage*.

C H A P T E R

2

Features of the Grapher Package

In this section, you will discover a high-level IBM® ILOG® Views package called the Grapher. This package includes powerful features dedicated to the graphic representation of hierarchical and interconnected information. This section contains information on the following:

- *Graph Management* - The first section introduces you to the graph management class `IlvGrapher`. This class is a natural extension of the manager concepts. It is based on the `IlvManager` class, and adds built-in mechanisms to handle interconnected graphic objects.
- *Grapher Links* - The second section explains the concept of *grapher link*s and how these entities are represented by a class hierarchy of customizable graphic objects.
- *Grapher Interactors* - The third section demonstrates how you can interact with a graph representation through several families of interactors.

Graph Management

This section describes the management of graphs in IBM ILOG Views. It is divided into two parts:

- *Description of the IlvGrapher Class*

Description of the IlvGrapher Class

Graphic objects representing graphs are stored in instances of the `IlvGrapher` class. This class derives from the `IlvManager` class and inherits all its features. The constructors of `IlvManager` (the base class) and `IlvGrapher` have the same parameters:

```
IlvGrapher(IlvDisplay*     display,
           int             layers    =  2,
           IlBoolean      useacc    =  IlTrue,
           IlvUShort       maxInList =  IlvMaxObjectsInList,
           IlvUShort       maxInNode =  IlvMaxObjectsInList);
```

In addition to the `IlvManager` concepts, the `IlvGrapher` class introduces a distinction between three types of graphic objects:

- **Nodes** - Nodes are the visual reference points in a hierarchy of information. A node is a graphic object—a subtype of the `IlvGraphic` class—that takes on a particular functionality when added to the grapher with the `IlvGrapher::addNode` method. This functionality allows links and nodes to stay connected when a node is moved.
- **Links** - Links are the visual representation of connections between nodes. A link is an instance of the `IlvLinkImage` class or one of its subclasses. It is added to the grapher with the `IlvGrapher::addLink` method. Since links can only exist between two existing nodes, you must create them with two graphic objects that are known as nodes by the grapher. You can use *ghost nodes* (added with the `IlvGrapher::addGhostNode` method) to create free-end links.
- **Ordinary graphic objects -** As is the case in a regular `IlvManager` instance, you can incorporate in your graph any `IlvGraphic` objects that represent neither nodes nor links.

The `IlvGrapher` class provides a set of member functions to manage links and nodes. You can, for example, replace a link with another one through a call to the `IlvGrapher::changeLink` method.

You can also transform a graphic object stored in the grapher into a node by calling the `IlvGrapher::makeNode` method. You can apply this method to a grapher link. This allows you to connect the link to other nodes. When dealing with a link that has a node behavior, you must make sure that there is no cycle in the geometric dependencies that govern the position of this link. Similarly, you can transform a graphic object into a grapher link with the `IlvGrapher::makeLink` method. The created link will be an instance of the `IlvLinkHandle` class, which is described in section *Grapher Links*.

Once objects are stored in an `IlvGrapher`, you can make a distinction between nodes, links, and ordinary graphic instances by using the `IlvGrapher::isNode` and `IlvGrapher::isLink` methods.

The IlvGrapher API also provides several methods to query the topology of your graph. For example, you can test whether two given nodes are connected by using the IlvGrapher::isLinkBetween method. You can also retrieve all the outgoing or incoming links of a node by using the IlvGrapher::getLinks method.

The sample code below shows how to use the IlvGrapher::mapLinks method to select all the outgoing links of a node:

```
static void SelectLink(IlvGraphic* g, IlvAny arg)
{
ILVCAST(IlvGrapher*,arg)->setSelected(g,IlTrue);
}

{
...
IlvGrapher* graph = ....;
IlvGraphic* node = ....; // The node being considered
//== Call the SelectLink function on all outgoing links of <node>
graph->mapLinks(node,SelectLink,graph,IlvLinkFrom);
...
}
```

Finally, the IlvGrapher class provides two predefined layout methods to arrange nodes in a vertical or horizontal tree structure. These layouts are implemented in the IlvGrapher::nodeXPretty and IlvGrapher::nodeYPretty methods.

An example showing how to create a simple grapher is provided in the <ILVHOME>/samples/grapher/simple directory. Also, you can refer to the *IBM ILOG Views Grapher Reference Manual* for more information on the member functions of the IlvGrapher class.

Loading and Saving Graph Descriptions

The IlvGrapher class reads graphs by using the IlvGraphInputFile class, and saves graphs by using the IlvGraphOutputFile class.

IlvGraphOutputFile

The IlvGraphOutputFile class is a subclass of IlvManagerOutputFile. In this subclass, the virtual method IlvGraphOutputFile::writeObject has been redefined to add specific information about each object before its description block. In our case, this information is the layer index, the type of the object (node, link, both, or an ordinary object), as well as the connection pins. Connection pins are described in section *Grapher Links*.

IlvGraphInputFile

The IlvGraphInputFile class is a subclass of IlvManagerInputFile. In this subclass the virtual method IlvGraphInputFile::readObject has been redefined to read the specific information written by the IlvGraphOutputFile::writeObject method.

Grapher Links

This section introduces the C++ classes that implement links in a grapher. These classes inherit the interface of the `IlvGraphic` class and add specific methods to handle the relationship between a link and its connected nodes. The following items are described:

- *Base Class for Links*
- *Predefined Grapher Links*
- *Creating a Custom Grapher link*
- *Connection Pins*

Base Class for Links

Figure 2.1 illustrates a straight link connecting two nodes:

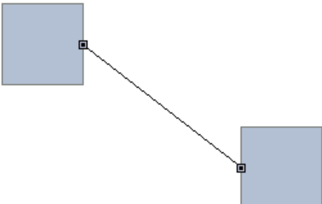


Figure 2.1 *Direct Link Between Two Nodes*

An `IlvLinkImage` instance is a graphic object that represents the connection between two nodes. By default, it is drawn as a straight line joining the two nodes. The constructor of the `IlvLinkImage` class is as follows:

```
IlvLinkImage(IlvDisplay* display,
             IlBoolean  oriented,
             IlvGraphic* from,
             IlvGraphic* to,
             IlvPalette* palette=0);
```

The `from` parameter is an object of type `IlvGraphic` that represents the start node of the link. The `to` parameter is an object of type `IlvGraphic` object that represents its end node. The `oriented` parameter specifies whether the link ends with an arrow-head.

Several member functions, prefixed by `set` and `get`, let you access these properties. For example, the end node can be accessed with the `IlvLinkImage::getTo` and `IlvLinkImage::setTo` methods. Similarly, you can change the oriented mode of the link with the `IlvLinkImage::setOriented` method.

Besides storing these properties, the purpose of the `IlvLinkImage` class is to:

- Compute the shape of the link as a function of its associated nodes and define how the link behaves when the geometry of the nodes changes. This task is carried out by the `IlvLinkImage::getLinkPoints` virtual method.
- Define how the link is drawn. This is done using the computed shape and is implemented in the virtual methods inherited from the `IlvGraphic` class.

Subclassing `IlvLinkImage` is useful when you want to create a link with a different behavior and/or drawing aspect. To change the behavior, overload the `IlvLinkImage::getLinkPoints` method:

```
virtual IlvPoint* getLinkPoints(IlUInt& count,
                                const IlvTransformer* t) const;
```

The returned array should not be deleted by the caller. You need to allocate this array on a common memory pool by using the `IlvPointPool` class. In this method, you can query the geometry of the start and end nodes to determine the points defining the shape of the link. There are two categories of such points:

- The end points of the link. These define where the link starts and ends.
- The intermediate points. These define the overall aspect of the link.

The `IlvLinkImage` class uses the `IlvLinkImage::computePoints` method to compute the location of the end points of the link:

```
virtual void computePoints(IlvPoint& src,
                           IlvPoint& dst,
                           const IlvTransformer* t = 0) const;
```

The default implementation first checks whether the link is associated with a connection pin on the nodes. (See section *Connection Pin Management Class* for more information.) If no connection pin is defined, the intersection of the link with the bounding boxes of the start and end nodes is computed. This is illustrated in Figure 2.2:

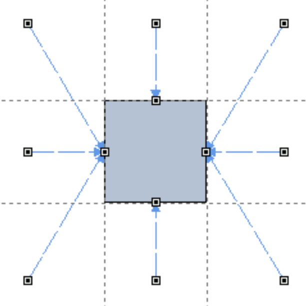


Figure 2.2 *End point Location When No Connection Pin is Defined*

Predefined Grapher Links

Predefined link classes are available in the grapher library. Each of these classes adds a specific behavior or drawing functionality to the `IlvLinkImage` base class. You can either use these classes as they are or subclass them to create customized links. The following classes are available:

- *IlvLinkHandle*
- *IlvLinkLabel*
- *IlvOneLinkImage*
- *IlvOneSplineLinkImage*
- *IlvDoubleLinkImage*
- *IlvDoubleSplineLinkImage*
- *IlvArcLinkImage*
- *IlvPolylineLinkImage*

IlvLinkHandle

The `IlvLinkHandle` class is an example of a link class where the shape and behavior of the link are directly inherited from `IlvLinkImage`, and where only the drawing of the link has been redefined.

This class lets you reference any type of graphic object to make it behave as a grapher link. Also, a graphic object can be referenced by several `IlvLinkHandle` instances. This allows you to create very lightweight links with complex shapes. Figure 2.3 illustrates an example of an `IlvLinkHandle` instance referencing a polygon:

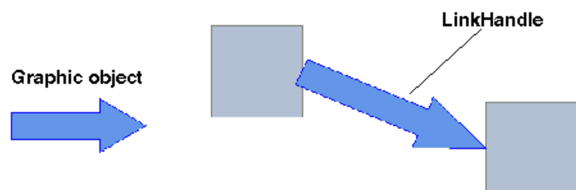


Figure 2.3 *Graphic Objects Used as a Link*

The constructor of this class is as follows:

```
IlvLinkHandle(IlvDisplay* display,
              IlvGraphic* object,
              IlvGraphic* from,
              IlvGraphic* to,
              IlvDim width = 0,
              IlBoolean owner = IlTrue
              IlvPalette*  palette=0);
```

Once added to the grapher, this instance will draw the graphic object `object` as a link between the nodes `from` and `to`, using the width `width`. The `owner` parameter describes the relationship between the handle and its referenced object. When a handle owns its referenced object, the handle is responsible for deleting this object. This means that you can safely share a referenced object as long as it is not owned by any of its handles.

An example showing how to use the `IlvLinkHandle` class is provided in the `<ILVHOME>/samples/grapher/linkhand` directory.

IlvLinkLabel

The `IlvLinkLabel` class also inherits the shape and behavior of the `IlvLinkImage` class. Links of the `IlvLinkLabel` type can be labelled with a user-defined character string.

This string can be specified by means of the `label` parameter of the constructor. It can also be specified once the link is created, by using the `IlvLinkLabel::setLabel` method.

Figure 2.4 shows two `IlvLinkLabel` objects:

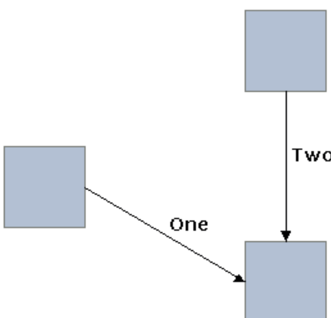


Figure 2.4 Labelled Links

IlvOneLinkImage

The `IlvOneLinkImage` class derives from the `IlvLinkImage` class and defines a new shape and a new behavior. Instances of this class are composed of two perpendicular lines, as illustrated in Figure 2.5:

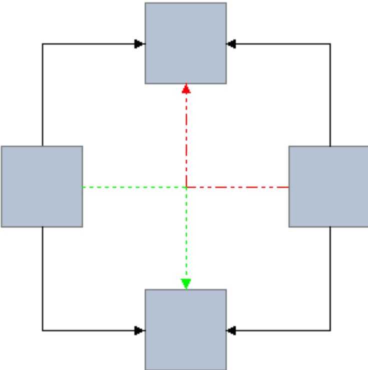


Figure 2.5 IlvOneLinkImage

The shape of the link depends on its *orientation* property, which indicates whether the link that leaves the `from` node starts out vertically (`IlvVerticalLink`) or horizontally (`IlvHorizontalLink`). This property can be specified in the constructor or it can be specified once the link is created, by using the `IlvOneLinkImage::setOrientation` method.

IlvOneSplineLinkImage

This class is a subclass of `IlvOneLinkImage` that draws the link as a spline:

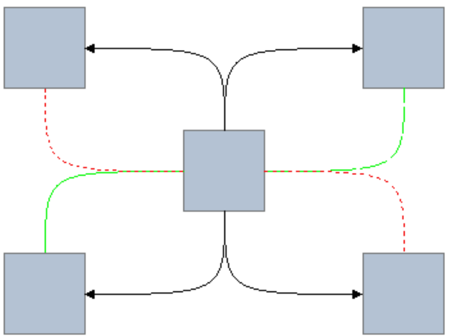


Figure 2.6 IlvOneSplineLinkImage

The position of the end points is similar to the one computed in the `IlvOneLinkImage` class. The two control points of the drawn spline are both at the intersection of the start and end tangents of the link. You can modify the position of the double-control point by using the `IlvOneSplineLinkImage::setControlPoint` method.

IlvDoubleLinkImage

The `IlvDoubleLinkImage` class derives from `IlvLinkImage` and defines a new shape and a new behavior. Instances of this class are composed of three connected lines intersecting at a 90° angle, as illustrated in Figure 2.7.

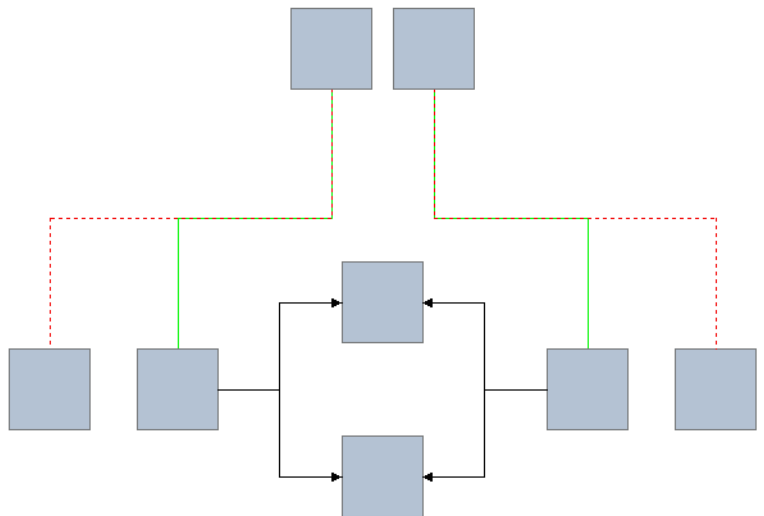


Figure 2.7 *IlvDoubleLinkImage*

The layout of the three segments follows two modes that are set with the `IlvDoubleLinkImage::setFixedOrientation` method:

- Automatic - The orientation of the segments depends on the vertical and horizontal separation between the two nodes. The middle segment takes the orientation of the largest separation.
- Fixed - The orientation of the link is fixed and specifies the direction (horizontal or vertical) the link takes upon leaving the starting node.

IlvDoubleSplineLinkImage

The `IlvDoubleSplineLinkImage` class is a subclass of `IlvDoubleLinkImage` that draws the links with smooth curves instead of straight segments, as shown in Figure 2.8. The behavior of these links is the same as in the `IlvDoubleLinkImage` class.

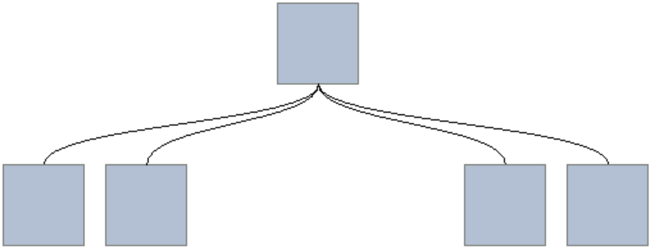


Figure 2.8 *IlvDoubleSplineLinkImage*

IlvArcLinkImage

The `IlvArcLinkImage` class is a subclass of `IlvLinkImage` that defines a new shape and a new behavior. Links of this type are drawn as an arc joining the two nodes, as shown in Figure 2.9:

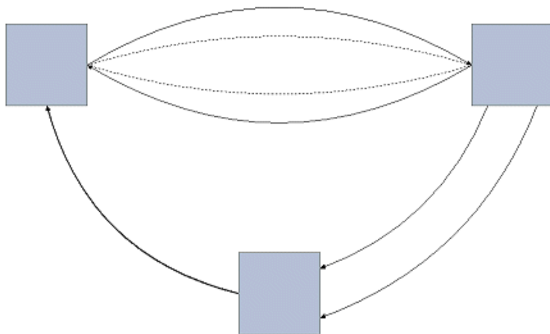


Figure 2.9 *IlvArcLinkImage Joining Three Nodes*

The arc is drawn as a spline with two control points. The distance between these control points and the segment joining the end points of the link (also called the *arc offset*) can be specified with one of the following:

- A fixed value, using the `IlvArcLinkImage::setFixedOffset` method,
- A value proportional to the length of the segment, using the `IlvArcLinkImage::setOffsetRatio` method.

This arc offset can take negative values, in which case the control points are located on the right of the oriented segment joining the start and end points. You can therefore connect two nodes with several links without any overlapping, by using different arc offsets.

IlvPolylineLinkImage

This class lets you dynamically define the intermediate points of a link. These points are stored in each `IlvPolylineLinkImage` instance and can be specified using several methods:

- `IlvPolylineLinkImage::setPoints`
- `IlvPolylineLinkImage::addPoints`
- `IlvPolylineLinkImage::removePoints`
- `IlvPolylineLinkImage::movePoint`

As with all link classes, the resulting shape is computed in the `IlvPolylineLinkImage::getLinkPoints` method. You can also specify whether the link is to be drawn with straight segments or with curves by calling the `IlvPolylineLinkImage::drawSpline` method. Figure 2.10 shows an example of the free-form links created by `IlvPolylineLinkImage` instances:

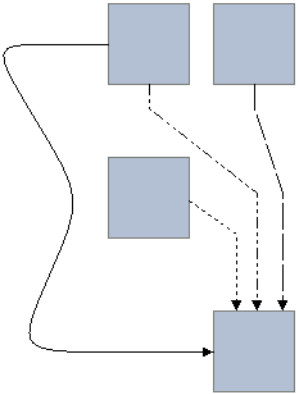


Figure 2.10 *IlvPolylineLinkImage*

Creating a Custom Grapher link

In this section, `IlvLinkImage` is subclassed to create a grapher link that meets the following specifications:

- The link is always drawn as a straight line between its two nodes.
- The start point is either defined by a connection pin or located at the center of the start node.

◆ The end point is such that the link stays perpendicular to the face of the end node closest to the start point. If this cannot be done, the end point is located on the closest corner of the node bounding box.

The link is drawn the same way as in the base class `IlvLinkImage`. Therefore, the corresponding methods inherited from `IlvGraphic` are left unchanged. Also, there are only two points defining the shape of the link (the two end points, and no intermediate points). There are two possibilities for defining the link: overloading the `IlvLinkImage::getLinkPoints` method or the `IlvLinkImage::computePoints` method. The second alternative has been chosen for this example:

```
void
MyLink::computePoints(IlvPoint& src,
                      IlvPoint& dst,
                      const IlvTransformer* t) const
{
    //== [1] ==
    IlvGrapherPin* pin = IlvGrapherPin::Get(getFrom());
    if (!pin || !pin->getLinkLocation(getFrom(),this,t,src)) {
        IlvRect bbox;
        getFrom()->boundingBox(bbox,t);
        src.move(bbox.centerx(),bbox.centery());
    }

    //== [2] ==
    IlvRect toBBox;
    getTo()->boundingBox(toBBox,t);
    if (src.x()<toBBox.x()) {
        if (src.y() < toBBox.y()) // Upper left quadrant
            dst.move(toBBox.x(),
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower left quadrant
            dst.move(toBBox.x(),
                     toBBox.y()+toBBox.h()-1);
        else // Left quadrant
            dst.move(toBBox.x(),
                     src.y());
    } else if (src.x()>=toBBox.right()) {

        if (src.y() < toBBox.y()) // Upper right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     toBBox.y()+toBBox.h()-1);
        else // Right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     src.y());
    } else {
        if (src.y() < toBBox.y()) // Upper quadrant
            dst.move(src.x(),
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower quadrant
            dst.move(src.x(),
                     toBBox.y()+toBBox.h()-1);
        else // src inside toBBox
```

```
            dst.move(toBBox.centerx(),toBBox.centery());
    }
}
```

In the first part (`[1]`) of the code, a verification is made to see whether the link is attached to a connection pin defined on its start node. If this is not the case, the center of the bounding box of this node is taken.

Once the location of the start point has been computed, the position of the start point with respect to the bounding box of the end node is verified (`[2]`). There are nine possible cases (the eight quadrants defined by `toBBox`, plus the case where the start point is inside `toBBox`), each defining a unique location.

Connection Pins

Connection pins allow you to control the exact location of link end points on grapher nodes. When a link is attached to a connection pin, the connecting point stays the same, regardless of the relative position of its start and end nodes.

The following items are described in this section:

- *Connection Pin Management Class*
- *An All-Purpose IlvGrapherPin Subclass*
- *Extending the IlvGrapherPin Class*

Connection Pin Management Class

The `IlvGrapherPin` abstract class is designed to handle a collection of connection pins. Its first purpose is to maintain the association between links and pins. To do so, pins are referenced by indexes. You can connect a link to a given connection pin with the `IlvGrapherPin::setPinIndex` method:

```
IlvLinkImage* link = …;
//== Recover the IlvGrapherPin instance associated with the starting node
IlvGrapherPin* pin  =  IlvGrapherPin::Get(link->getFrom());
//== Connect the link to the pin whose index is 0
pin->setPinIndex(link,0,IlTrue);
```

Likewise, you can recover the index of the connection pin to which a link is attached, by using the `IlvGrapherPin::getPinIndex` method.

The second purpose of the `IlvGrapherPin` class is to provide an interface to query the coordinates of the connecting points available for a given node. Each concrete subclass must provide an implementation for the `IlvGrapherPin::getCardinal` and `IlvGrapherPin::getLocation` methods:

```
virtual IlUInt getCardinal(const IlvGraphic* node,
                           const IlvTransformer* t) const;
```

This method returns the number of connection pins handled by the instance for the specified node `node` when displayed with the transformer `t`.

```
virtual IlBoolean getLocation(IlUInt pinIndex,
                              const IlvGraphic* node,
                              const IlvTransformer* t,
                              IlvPoint& where) const;
```

This method returns, in the `where` parameter, the coordinates of the connection pin specified by the index `pinIndex` on the node `node`, when displayed with the transformer `t`.

Other methods of this interface (`IlvGrapherPin::getClosest`, `IlvGrapherPin::getLinkLocation`, and so on) have a default implementation that can be overloaded. For example, the `getClosest` method considers all available connection pins and uses the `getLocation` method. You can change this method to:

- provide a faster implementation (`getLocation` may contain computations that can be done only once in `getClosest`),
- return the first unused pin instead of the closest one in terms of distance.

An All-Purpose IlvGrapherPin Subclass

The `IlvGenericPin` class is a predefined concrete subclass of `IlvGrapherPin` that makes it possible to dynamically define the connection pins on a node. New connection pins are specified by their desired location on the node when this node is displayed through a given transformer. Once this position is stored, the `IlvGenericPin` class will use the shape of the object to accurately locate the connecting point regardless of the applied transformer.

Here is an example of how to use this class to add connection pins on the four corners of a node bounding box:

```
IlvGraphic* node = ...;
//== Create an empty instance of IlvGenericPin
IlvGenericPin* pin = new IlvGenericPin();
//== Add the four connecting points
IlvRect bbox;
node->boundingBox(bbox,0);
pin->addPin(node,IlvPoint(bbox.x(),bbox.y()),0);
pin->addPin(node,IlvPoint(bbox.x()+bbox.w()-1,bbox.y()),0);
pin->addPin(node,IlvPoint(bbox.x()+bbox.w()-1,bbox.y()+bbox.h()-1),0);
pin->addPin(node,IlvPoint(bbox.x(),bbox.y()+bbox.h()-1),0);
//== Attach the IlvGenericPin instance to the node
pin->set(node);
```

> ***Note:*** *The points in this example are given in the object coordinate system when no transformer is applied.*

Extending the IlvGrapherPin Class

An example of a concrete `IlvGrapherPin` subclass that handles a single connection pin located at the center of a node bounding box is presented here. This class, called `CenterPin`, is declared as follows:

```
#include <ilviews/grapher/pin.h>

class CenterPin
: public IlvGrapherPin
{
public:
    CenterPin() {}

    virtual IlUInt getCardinal(const IlvGraphic*,
                               const IlvTransformer*) const;

    virtual IlBoolean getLocation(IlUInt,
                                  const IlvGraphic*,
                                  const IlvTransformer* t,
                                  IlvPoint&) const;
    DeclarePropertyInfoRO();
    DeclarePropertyIOConstructors(CenterPin);
};
```

The constructor of the `CenterPin` class does nothing since this class does not store any information. The `DeclarePropertyInfoRO` and `DeclarePropertyIOConstructors` macros are used to make the `CenterPin` class persistent. Only the `getCardinal` and `getLocation` methods are overloaded since the implementation of the other `IlvGrapherPin` methods does not need to be changed. The source file for the `CenterPin` class defines the following methods:

```
#include <centerpin.h>

// --------------------------------------------------------------------------
// - IO Constructors
CenterPin::CenterPin(IlvInputFile& input, IlvSymbol* s)
: IlvGrapherPin(input, s) {}

CenterPin::CenterPin(const CenterPin& src)
: IlvGrapherPin(src) {}
// --------------------------------------------------------------------------
IlUInt
CenterPin::getCardinal(const IlvGraphic*,
                       const IlvTransformer*) const
{
    return 1;
}

// --------------------------------------------------------------------------
IlBoolean
CenterPin::getLocation(IlUInt,
                       const IlvGraphic* node,
                       const IlvTransformer* t,
                       IlvPoint& where) const
{
```

```
    IlvRect bbox;
    node->boundingBox(bbox, t);
    where.move(bbox.centerx(), bbox.centery());
    return IlTrue;
}

// --------------------------------------------------------------------------
// - Macros to register the class and make it persistent
IlvPredefinedPropertyIOMembers(CenterPin)
IlvRegisterPropertyClass(CenterPin, IlvGrapherPin);
```

The implementation of the `getCardinal` method is straightforward and returns 1 for any node and transformer. The `getLocation` method simply queries the transformed bounding box of the node and returns its center. (The index of the connection pin is not used since this class defines only one connection pin.) The declaration of the `CenterPin` class is provided in the file `<ILVHOME>/samples/grapher/include/centerpin.h`. Its implementation can be found in the file `<ILVHOME>/samples/grapher/src/centerpin.cpp`.

Grapher Interactors

The `IlvManager` class provides a wide range of interactors that are used to create objects and change their shape. The `IlvGrapher` class contains specific interactors designed to create new nodes and links and change the way they are connected:

- *Selection Interactor*
- *Creating Nodes*
- *Creating Links*
- *Editing Connection Pins*
- *Editing Links*

Selection Interactor

The `IlvGraphSelectInteractor` class derives from the `IlvSelectInteractor` class. It contains additional member functions used to manage the drawing of ghost images for links attached to nodes that are moved or enlarged. This class has the following constructor:

```
IlvGraphSelectInteractor(IlvManager* manager, IlvView* view);
```

This constructor initializes a new instance of the `IlvGraphSelectInteractor` class that lets you select individual objects or groups of objects in the view `view` connected to the manager `manager`. This manager is assumed to be an instance of the `IlvGrapher` class.

Creating Nodes

The IlvMakeNodeInteractor class is the base class for interactors that allow the user to interactively create nodes in a grapher. Instances of this class must be attached to a grapher and one of its connected views, as shown here:

```
IlvGrapher* graph = ...;
IlvView* view = ...;
IlvMakeNodeInteractor * inter = new IlvMakeNodeInteractor(graph, view);
graph->setInteractor(inter);
```

To create a node, drag a rectangular region in the working view. There are two ways to specify what type of graphic object is created:

- Subtype the IlvMakeNodeInteractor class and overload its IlvMakeNodeInteractor::createNode method.
- Subtype the IlvMakeNodeInteractorFactory class and overload its IlvMakeNodeInteractorFactory::createNode method. You can associate a node factory with an interactor by using the IlvMakeNodeInteractor::setFactory method.

The grapher library provides predefined subclasses of IlvMakeNodeInteractor:

- IlvMakeShadowNodeInteractor - This interactor creates instances of the IlvShadowLabel class and stores them as nodes in the grapher.
- IlvMakeReliefNodeInteractor - This interactor creates instances of the IlvReliefLabel class and stores them as nodes in the grapher.

Creating Links

The IlvMakeLinkInteractor class is the base class for interactors that allow the user to interactively connect nodes in a grapher. Its constructor is as follows:

```
IlvMakeLinkInteractor(IlvManager* manager,
                      IlvView* view,
                      IlBoolean oriented = IlTrue);
```

The oriented parameter specifies whether created links are oriented. An example of how to create an interactor of this type and connect it to a grapher and one of its view is presented here:

```
IlvGrapher* graph = ...;
IlvView* view = graph->getFirstView();
IlvMakeLinkInteractor * inter = new IlvMakeLinkInteractor(graph, view);
graph->setInteractor(inter);
```

To connect two nodes, perform the following steps:

1. Click the starting node. This node is highlighted if it is considered valid by the interactor.

2. Drag the mouse until it is positioned over the ending node. If this node is valid, it is also highlighted.
3. Release the mouse button to create the link.

You can control which node is valid by overloading the `IlvMakeLinkInteractor::acceptFrom` and `IlvMakeLinkInteractor::acceptTo` methods. There are two ways of specifying what type of link should be created:

- Subtype the `IlvMakeLinkInteractor` class and overload its `IlvMakeLinkInteractor::createLink` method.
- Subtype the `IlvMakeLinkInteractorFactory` class and overload its `IlvMakeLinkInteractorFactory::createLink` method. You can associate a link factory with an interactor by using the `IlvMakeLinkInteractor::setFactory` method.

The Grapher library provides several predefined subclasses of `IlvMakeLinkInteractor`:

- `IlvMakeLinkImageInteractor` - This class is used to create a link of type `IlvLinkImage`.
- `IlvMakeLabelLinkImageInteractor` - This class is used to create a link of type `IlvLinkLabel`.
- `IlvMakeOneLinkImageInteractor` - This class is used to create a link of type `IlvOneLinkImage`.
- `IlvMakeOneSplineLinkImageInteractor` - This class is used to create a link of type `IlvOneSplineLinkImage`.
- `IlvMakeDoubleLinkImageInteractor` - This class is used to create a link of type `IlvDoubleLinkImage`.
- `IlvMakeDoubleSplineLinkImageInteractor` - This class is used to create a link of type `IlvDoubleSplineLinkImage`.

Creating Polyline Links

The `IlvMakePolyLinkInteractor` class is a special kind of interactor that does not derive from `IlvMakeLinkInteractor`.

This interactor is used to create links whose intermediate points can be explicitly defined. It lets you control the shape drawn by the user by means of the `IlvMakePolyLinkInteractor::accept` method:

```
virtual IlBoolean accept(IlvPoint& point);
```

By overloading this method, you can add specific constraints on the position of the intermediate points of the link. Once these points have been defined, the link is created with the `IlvMakePolyLinkInteractor::makeLink` method, which must be defined in subclasses to return the appropriate link instance. The grapher library provides one

predefined subclass, IlvMakePolylineLinkInteractor, which is used to create links of the IlvPolylineLinkImage type.

Editing Connection Pins

The IlvPinEditorInteractor class lets the user interactively edit the connection pins of a grapher node. When this interactor is active, selecting a node will highlight its connection pins, as shown in Figure 2.11:

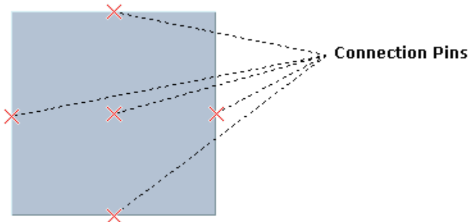


Figure 2.11 *Highlighted Connection Pins*

Once a grapher node is selected, you can:

- Add a new connection pin by clicking inside the node.
- Remove a connection pin. To do this, select the pin with the mouse and press the Delete key.
- Move an existing connection pin. To do this, select the pin with the mouse and drag it to its desired location.
- Connect and disconnect links to or from a pin. To do this, first select a connection pin, and then click the considered link.

> ***Note:*** *If the working node is already associated with a pin management object, this object must be of the* IlvGenericPin *type. If the node does not define any connection pin, then an* IlvGenericPin *instance is automatically created.*

Editing Links

When a link is selected, its selection object draws handles that you can use to change its shape or edit the way it is connected. Figure 2.12 shows a link that has been selected:

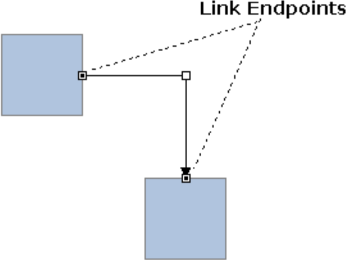


Figure 2.12 A Selected Link

An end point handle can be dragged to:

- Change the connection pin to which the link is attached. When the handle is dragged near a connection pin, the pin is highlighted and the link uses its position to compute the location of its end point.
- Connect the link to another node.

The intermediate point handles can be used to edit the shape of the link. The kind of interaction allowed by these handles depends on the kind of link being edited.

> ***Note:*** *Link editing can be turned off by using the* `IlvGrapher::setLinksEditable` *method. When an* `IlvGrapher` *instance is created, link editing is disabled by default.*

Part III

Prototypes

Part III explains the concepts behind Business Graphic Objects and shows how to create and use prototypes with both containers and managers. This part consists of the following chapters:

- ***Chapter 1, Introducing the Prototypes Package*** introduces the concepts of prototypes.
- ***Chapter 3, The User Interface and Commands*** describes the main parts of IBM ILOG Views Studio with the Prototypes extension.
- ***Chapter 2, Using IBM ILOG Views Studio to Create BGOs*** explains how to use IBM ILOG Views Studio to create your prototypes by composing graphic objects and assigning behaviors to them.
- ***Chapter 4, Using Prototypes in C++ Applications*** describes the classes and methods used to manipulate prototypes and shows how to structure your application to benefit fully from BGOs. It then shows how to use prototypes created in IBM ILOG Views Studio in your C++ applications.

- ***Chapter 5, Predefined Accessors*** lists the behaviors that are predefined in the Prototypes library.

> ***Note:*** *The IBM ILOG Views Prototype package is available only if you have purchased the IBM ILOG Views 2D Graphics Professional product.*

CHAPTER

1

Introducing the Prototypes Package

The Prototypes package lets you create custom domain-specific graphic objects called *Business Graphic Objects (BGOs)*. These objects are created interactively, without writing C++ code, using the Prototypes extension of IBM ILOG Views Studio.

This section introduces the concepts of BGOs and explains the classes and methods used to manipulate the prototypes created with IBM ILOG Views Studio.

An Overview of the Prototypes Package

This section provides an overview of how to use the Prototypes package of IBM® ILOG® Views to create BGOs. Because IBM ILOG Views BGOs are based on the prototype design pattern, they are often referred to as prototypes.

The following items are described in this section as an introduction to the Prototypes package:

- *Business Graphic Objects*
- *Creating BGOs Using the Prototypes Extension of IBM ILOG Views Studio*
- *Using Prototypes in Applications*
- *When Should You Use Prototypes?*

- *The Prototype Design Pattern*
- *Specifying Graphic and Interactive Behavior Using Accessors*

Business Graphic Objects

Application developers often need to define custom graphic objects to represent domain-specific application objects that the user is able to interact with. The IBM® ILOG® Views Prototypes package provides a simple and efficient solution for building such business graphic objects (BGOs). BGOs are created using the Prototypes extension of IBM ILOG Views Studio. Creating a BGO requires no coding. It is created by performing three basic steps:

1. Define the application interface of the BGO as a set of typed attributes that represent the domain of your object. For example, a boiler object representing a power plant boiler can have Temperature, Capacity, Level, Input valve, and Output valve attributes.
2. Define the look of your objects using basic IBM ILOG Views graphic objects, such as lines, text, and images. You can also include other BGOs to build structured objects. For example, the boiler object could be represented by a rectangle, the temperature and level by gauges inside the rectangle, and input and output valves by toggle buttons inside the rectangle.
3. Attach behaviors to your graphic objects to define how they should represent the state of an application object and how they should react to user events. You can dynamically change the attributes of a shape, animate the object, and connect BGOs together to reflect the state of the objects in the user interface. For example, attaching a Fill behavior to the Level attribute ensures that the level of the boiler is kept synchronized with its graphic representation.

You can then create instances of your BGOs and use them in managers or containers just as you would do with basic IBM ILOG Views graphic objects. You can link application objects to their corresponding BGO. The display, synchronization, and user interaction is handled by the Prototypes package. You can edit and modify a BGO at any time: its instances will be automatically updated.

Creating a powerful, direct-manipulation interface for domain-specific objects becomes as easy as creating a form-based interface for the same objects, but the resulting interface is much more appealing and explicit to the user.

Figure 1.1 shows examples of application panels built with prototypes.

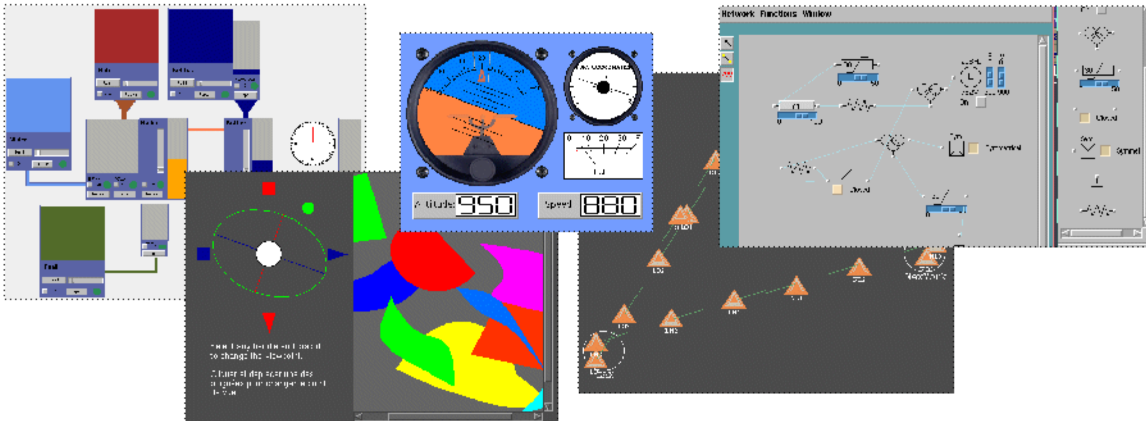


Figure 1.1 *Examples of Prototype Applications*

Creating BGOs Using the Prototypes Extension of IBM ILOG Views Studio

When you use IBM ILOG Views Studio to create your BGOs, you can:

- Design the graphic appearance of your BGOs by assembling basic graphic objects.
- Define the graphic behavior and interactive aspects of your BGOs by attaching predefined behaviors to them, or by writing scripts.
- Store the BGOs in libraries as *prototype* objects that can be reused, modified, and instantiated in panels. Since BGOs are mostly used as prototypes, the terms prototype and BGOs are used interchangeably.
- Add instances of your prototypes to managers or containers.
- Test the behavior of your prototypes and the panels that contain them.

All these operations are performed in WYSIWYG (what you see is what you get) mode without coding in C++.

Using Prototypes in Applications

You can load IBM® ILOG® Views files containing instances of your prototypes into a manager or a container the same way you load files containing basic IBM ILOG Views graphic objects. You can also create instances of prototypes, attach them to application objects, and place them in managers or containers.

Prototypes are not subclasses of `IlvGraphic`. They are groups of graphic objects contained in an object of the `IlvGroup` class. The definition of a prototype is stored in a file so you do not need to recompile your application if you modify a prototype.

To place prototype instances in an `IlvManager` or an `IlvContainer` you must embed them in a specific subclass of `IlvGraphic` called `IlvProtoGraphic`. When you use IBM ILOG Views Studio to place the prototypes in the manager or container, IBM ILOG Views Studio creates the encapsulating `IlvProtoGraphic` for you. You can manipulate `IlvProtoGraphic` the same way as an `IlvGraphic`. You can use the `IlvGroupHolder` class to retrieve the prototype instances of a given view (container or manager) and modify the properties of your prototype instances according to the application values you want to display.

When Should You Use Prototypes?

To define a BGO, you can either use prototypes or write the C++ code for a subclass of `IlvGraphic` using direct calls to the IBM ILOG Views methods to draw your object. The use of prototypes is therefore an alternative to direct coding.

The prototype approach has the following advantages:

- Very short development time that permits an iterative GUI design process.
- Easy maintenance and debugging, since there is a clear separation between the implementation of the application and the implementation of the user interface.
- Complete integration in ILOG Views Studio. The user interface designer draws instead of programming.
- Few C++ programming skills required.

As a result, the task of designing the graphical appearance of your objects can be delegated to non-programmers. For example, graphic designers may be more suited to the task and will find in IBM ILOG Views Studio a drawing program comparable to the graphic tools they are accustomed to.

Prototypes have been designed and implemented with a strong emphasis on efficiency. Although prototypes may not always be as efficient as direct C++ coding (because they are based on composition rather than derivation), applications can create thousands of prototype instances without encountering performance problems.

The Prototype Design Pattern

The process of creating BGOs is based on the prototype design pattern. You can group basic objects and use the group as a model (or prototype) from which you can create clones (or instances). When the prototype is modified, all its instances are automatically updated with this modification.

Using the prototype design pattern, it is possible to create complex graphic objects using a WYSIWYG editor and to use the objects immediately to build application panels.

Specifying Graphic and Interactive Behavior Using Accessors

BGOs define a set of public attributes. These attributes correspond to the application programming interface of the BGO. You can change the appearance of the BGO by setting its attributes to given values. You can also query any of these attributes at any time.

You can attach several behaviors to each of these attributes. A behavior defines a side effect that is executed each time the attribute is changed or queried. For example, you can attach a Condition behavior to a Temperature attribute. Each time the temperature is changed, the condition is evaluated and the graphic appearance of the object changes. The Condition behavior can set the color of an object to red if the temperature is above a predefined threshold. You can also attach interactive behaviors to your BGO— for example, you can specify that the temperature should be adjusted when the user clicks on the thermometer.

Attributes and behaviors are implemented by means of accessors (objects of the class `IlvAccessor`). Accessors can be attached to graphic objects and can:

- Store attribute values
- Perform side effects
- Track user events

The accessor mechanism allows you to define complex behaviors. You can combine accessors to re-create the logic of an entire application. However, it is strongly recommended that you use the accessor mechanism only to specify the graphic and interactive behaviors of your objects. Do not use the accessor mechanism to implement features of the application domain. By doing this, you maintain the sound modular aspects of your program.

Taken as a whole, the accessors of a BGO define a *data flow* graphic program. Data flow programming is as powerful as the more classical *control flow* model used in programming languages such as C++ or Java. However, data flow programming is better adapted to the definition of small, graphic oriented programs.

To facilitate the definition of complex graphic behaviors, the Script accessor allows you to define graphic or interactive behavior as an IBM ILOG Script program. This allows more complex computations to be performed and gives access to the entire suite of IBM ILOG Script features.

C H A P T E R

2

Using IBM ILOG Views Studio to Create BGOs

This chapter describes how the Prototypes extension lets you create composite graphic objects and assign them an application interface, a graphic behavior, and an interactive behavior through interactive, point-and-click editing. These graphic objects can then be linked to domain-specific objects following the application interface, providing full WYSIWYG, direct-manipulation editing of the domain objects.

You can find information on the following topics:

- *Creating and Using Prototypes*
- *Loading and Saving Prototype Libraries*
- *Creating and Editing Prototype Instances in Panels*
- *Connecting Prototype Instances*

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

Creating and Using Prototypes

The following topics related to creating and using prototypes are presented in this section:

- *Creating a Prototype Library*
- *Creating a Prototype*
- *Defining the Attributes*
- *Drawing the Prototype*
- *Defining Graphic Behaviors*
- *Defining Interactive Behaviors*
- *Testing Your Prototype*
- *Saving a Prototype*

Creating a Prototype Library

You will probably want to create your BGOs in libraries so that you can retrieve and manipulate them all together.

To create a new prototype library, do the following:

1. From the File menu in the Main window, choose the command New > Prototype Library.

   A file selector appears.

2. Select a directory for which you have write permission and enter the name of the new library (it must have a `.ipl` extension). Click Save.

   A new page, corresponding to the library you have just created, appears in the Palettes panel.

Creating a Prototype

These are the tasks involved in creating a prototype:

- Defining the attributes of your prototype in the Interface page of the Group Inspector panel.
- Drawing the graphic elements that make up the prototype in the Prototype buffer window.
- Defining the graphic behavior of the prototype using the Group Inspector panel
- Defining the interactive behavior of the prototype using the Group Inspector panel.

These tasks can be interleaved: at any point you can add, edit, or remove attributes, graphic elements, or behaviors of the prototype.

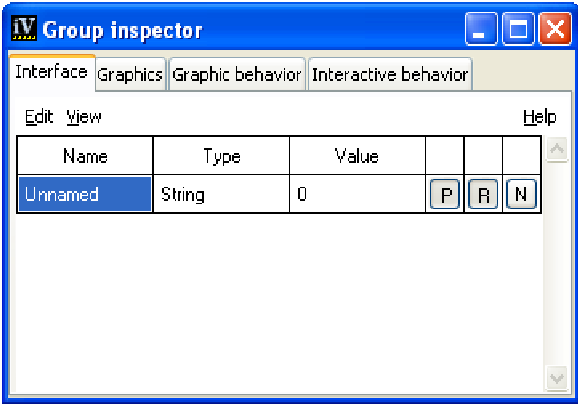
Your prototype is created in a prototype buffer window. Before beginning these tasks:

1. Open a Prototype buffer window by selecting New > Prototype from the File menu.
2. Display the Group Inspector panel by selecting the Group Inspector from the Tools menu.

Defining the Attributes

Use the following procedure to define and edit the external attributes (or "properties") of your prototype or group. (Properties determine how you will access your prototype or group from your application or from other objects.)

1. Open the Interface page of the Group Inspector Panel. This page allows you to define a set of attributes, giving each of them a type and a default value:



2. Choose Edit > New Attribute or Ctrl+N to add an attribute.

   A new row "Unnamed" appears in the table.

3. To specify the name of the attribute, click the box Unnamed. Enter a name for this attribute. This name must be unique to avoid ambiguities; it is used to access the behavior of this attribute.

4. To specify the type of the attribute, click twice on the adjacent Type combo box (or press F2 if using only the keyboard). Select the pull-down menu, which will let you specify a type.

   All attributes are typed: each type indicates the kind of values that can be assigned to the attribute, which helps determine its meaning. The types available are:

- **Value** - The attribute holds a value that can be set or queried directly (a string, a color, an integer, and so on). When you set an attribute to a given value type, the combo box will display this type directly.
- **Reference** - A reference to another internal attribute of the group. For instance, creating an attribute named "temperature", and having it reference the "value" attribute of a "slider" graphic object allows you to access the "value" internal attribute of the slider under the more appropriate name of "temperature", which is helpful if the group is to represent a thermometer. This is equivalent to a pointer or an alias in a programming language. When you have set the type to reference, the referenced attribute, prefixed with “^”, appears in the combo box that describes the attribute type.
- **Grouping of attributes** - All subattributes in the group bearing the name of the attribute will be addressed collectively and assigned the same value. For instance, creating an attribute named "foreground" and giving it the type "group" creates an attribute that will set the foreground of all objects contained in the group to the same value.
- **Script** - A script is executed. This script should return a value, which defines the attribute. Use the Behavior page to change the name of the function that defines the value. The name of the function, followed by “()”, appears when you choose this type of attribute.
- **NoType** - Some attributes can be purely functional, and therefore untyped.

> ***Note:*** *If an attribute has neither a type nor a behavior, it cannot exist. Therefore, setting a type of "none" to an attribute after creating it is equivalent to deleting the attribute.*

5. Enter a default value for the attribute in the Value column of the attribute.
6. To set other parameters of the attribute, use the buttons on the right side of the page. When the button is released, the property is set:
   - **Public** (button P in the inspector) - The attribute is visible by outside objects. Attributes are public by default, but you can hide those attributes that are only used to perform internal computations.
   - **Persistent** (button R in the inspector) - The attribute value will be stored when the group is saved, allowing the last value set by the user to be maintained. By default, attributes are persistent. To optimize reading and writing, or to always restore the attributes to the original state of the prototype when a file is read, you can set them to non-persistent.
   - **Notifying** (button N in the inspector) - When this is set, the attribute can notify other attributes that it has changed its value and, therefore, enable other attributes to update themselves. See section *Notify*.

7. Repeat steps 2 through 6 for each attribute you wish to add to provide a description of the interface for your prototype.

Using the Edit Menu on the Interface Page

When specifying the interface of your prototype, you can also:

- Import the interface of another Prototype to add predefined attributes and behaviors from that prototype:

  Select Edit > Delegate To Prototype and choose from the available list the prototype whose attributes you want to inherit.

  A new attribute is created with the inherited attributes shown grayed-out in the table. You cannot directly edit these inherited attributes, but you can reference them through other attributes.

  Some inherited attributes may already reference other attributes or graphic nodes, and therefore you may find that not every prototype can be imported into another prototype.

- Order the attributes:

  Select an attribute and choose Edit > Move Item Up or Edit > Move Item Down.

- Delete an attribute:

  Select the attribute and choose Edit > Delete.

- Cut/Copy/Paste: You can copy or cut a whole attribute and its behavior by selecting the first line of an attributes tree and selecting Edit > Copy or Edit > Cut. You can paste the content of the attribute's clipboard by first selecting a line where you want the attribute to be inserted, and then selecting Edit > Paste.

Using the View Menu on the Interface Page

This menu on the Interface page presents alternative views of the attributes of your group or prototype, and allows you to select which types of attributes you want to edit for a given group or prototype:

- Interface - Lets you access and edit all the attributes defined for the group or prototype. This is the default presentation.
- Public Attributes - Shows only the public attributes of the prototype, those that can be seen by other objects and by the application.
- Modified Values - Lists the values of a prototype instance that differ from its prototype. These values will be saved together with the prototype instance.
- All Values - Lists all the prototype values and subvalues. These values can be modified, but this does not mean that the modifications will be saved with the prototype if some other behaviors override the new settings.

Drawing the Prototype

Define the graphic presentation of your prototype using the Prototype buffer window:

- You can drag and drop graphic objects into the buffer and use the editing modes to create lines, polygons, and so on: a Prototype buffer window has all the properties of the IBM ILOG Views Studio 2D Graphics buffer window.
- As you draw your prototype, you can see its structure in the Graphics page of the Group Inspector panel, shown in addition to the Main window. Figure 2.1 shows an example of this. The list of graphic nodes appears organized from bottom to top. As you add graphic objects to the prototype, the tree structure is updated. You can select graphic nodes either directly in the Prototypes buffer window (as you do in IBM ILOG Views Studio) or in the tree that appears in the Group Inspector panel.

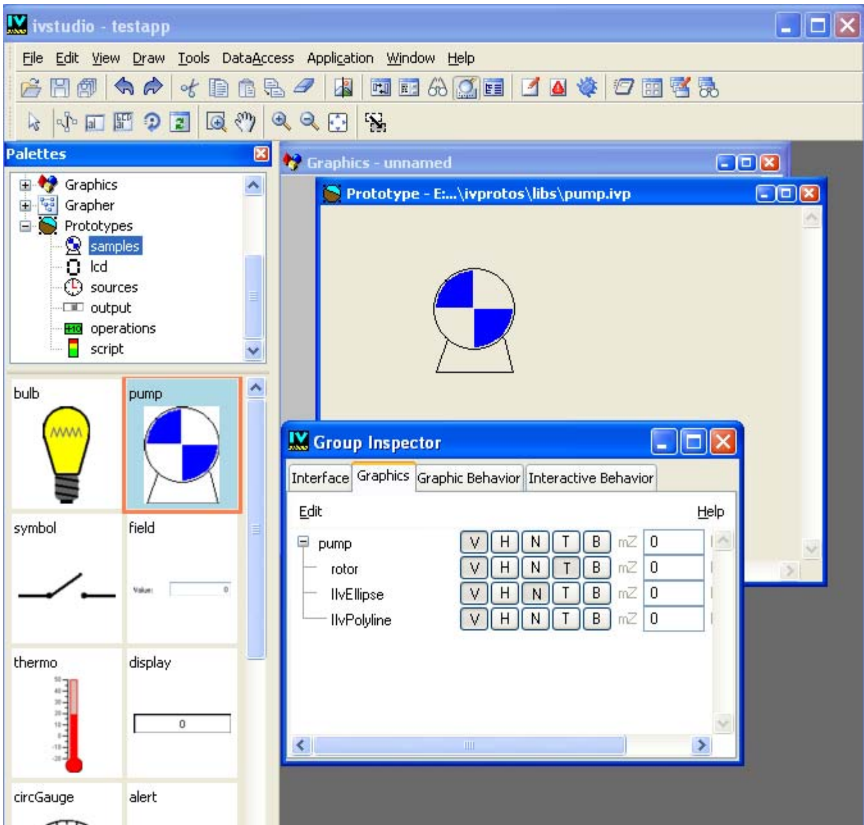


Figure 2.1 The Graphics Notebook Page of the Group Inspector Panel

Editing Prototype Nodes

You can use the fields on the Graphics page of the Group Inspector (see Figure 2.2) to change a number of properties associated with the elements (or nodes) in your prototypes:

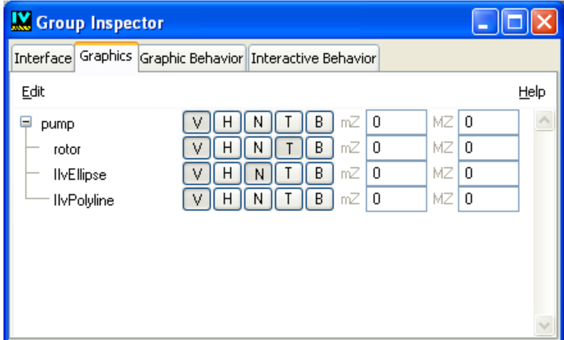


Figure 2.2 *The Graphics Notebook Page of the Group Inspector Panel*

- If the selected node is a graphic node, the properties only apply to this particular node.
- If the selected node is a group node—that is, the root node of the prototype, a subgroup of the prototype, or a prototype instance—the properties apply to all the child graphic nodes of the selected group.

The following table describes the fields found on the Graphics notebook page:

Table 2.1 *Fields of the Graphics Notebook Page of the Group Inspector Panel*

Field	**Description**
Node name	This text field is used to change the name of the node. You can also use the Name field of the Generic Inspector in the IBM ILOG Views Studio Main window. ***Note:*** *Nodes should contain only alpha-numeric characters (A-Z, a-z, 0-9).*
(V) Visible	This toggle controls the visibility of the graphic object in the prototype.
(H) Hidden in application	If this button is set, the selected graphic object is visible only while editing the prototype or its instances in IBM ILOG Views Studio. The object is hidden in the final application. This property can be used to create intermediate “computing” prototype instances such as those of the “operations” prototype library.

Table 2.1 Fields of the Graphics Notebook Page of the Group Inspector Panel (Continued)

Field	Description
(N) Grapher node	If this button is set, the graphic object is added as a grapher node when the prototype is instantiated in a grapher. This allows you to use prototype instances as grapher nodes. (This button is deprecated and included for compatibility reasons.)
(T) Transformed	This button controls whether a transformer is associated with the graphic node to ensure that the graphic object can be transformed arbitrarily without distortions. Without a local transformer, some IBM ILOG Views objects lose their original geometry when they are resized. Using a local transformer ensures that the geometry of objects is not modified by geometric transformations. On the other hand, using a local transformer consumes more memory. If you select this button, remember that you must use the standard Selection mode to inspect the graphic object of the node. If you use the Group Selection mode, the selected object is an instance of a subclass of `IlvTransformedGraphic` and cannot be inspected.
(B) Bounded Size	If set, this flag restricts the zoomability of the objects. Setting this flag and leaving mZ and MZ to 0 is equivalent to setting both of them to 1. It is, however, more efficient. If Bounded Size is set, and mZ or MZ are not 0, this flag makes the objects disappear if the zoom factor of the view of the instance is greater than MZ or less than mZ.
(mZ) Min. Zoom	If not zero, this attribute limits the minimum size an object can have. When the scaling factor of the view holding the object is below this value, the object does not get any larger. If Min. Zoom and Max. Zoom are set to the same value, the object never grows or shrinks in size. If they are set to 1, they stay at the size at which they were created.
(MZ) Max. Zoom	If not zero, this attribute limits the maximum size an object can have. When the scaling factor of the view holding the object is above this value, the object does not get any larger.

Structuring Prototype Nodes

To structure your prototype, you can group graphic nodes into subgroups. This may be useful when you define your prototype accessors; for example, when you want to rotate a group of objects or change their color. To create a subgroup:

1. In the Prototype buffer window, select the graphic objects you want to group.

2. From the Draw menu select Group.

The objects are grouped into an instance of the class `IlvGroup` and a subgroup node is created in the prototype. The node tree shows the structure change.

Using the Group Selection mode, subgroups can be selected and moved as a whole. This mode shows the selected group by drawing a dashed-line frame around the group. You are still able to select individual graphic objects inside subgroups with the standard Selection mode.

To include instances of other prototypes in the prototype you are editing:

1. If not already open, activate the Prototypes palette by choosing Palettes from the Tools menu.

2. Select the desired prototype library.

3. Drag and drop the prototype into the Prototype buffer window. The Nodes page of the Prototype Inspector will show a new node, similar to a subgroup node, for the prototype instance.

Having added objects, you may return to the Interface page to define new attributes that reference the internal nodes, or go to the Behavior page to define dynamic behaviors for the prototype.

Defining Graphic Behaviors

Define the graphic behaviors of your prototype using the Behavior page of the Group Inspector (see Figure 2.3). The graphic behaviors determine how the modification of an attribute affects the visual aspect of your prototype. For each attribute, you can add behaviors: these are instructions that will be performed each time the value is modified.

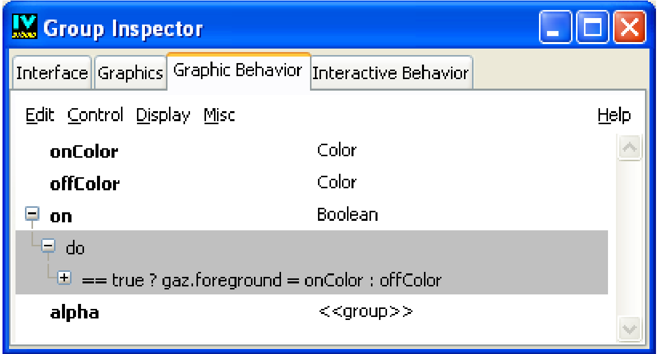


Figure 2.3 The Behavior Notebook Page of the Group Inspector

To add a behavior:

1. Select an attribute in the list.

2. From the Control, Display, or Misc menus, select a behavior to be added:

   - Control behaviors enable the change of one attribute to trigger changes of other attributes, conditionally or not. For instance, if you want a thermometer to appear red when the temperature is above a given threshold, add a Condition accessor on the temperature attribute that assigns red to the foreground value of the gauge.
   - Display behaviors enable you to change graphic properties of objects, such as rotation, zoom, and visibility, or perform animations of objects.

In addition, you can have attributes notify others of their changes, so that the graphic appearance of the group or prototype can be fully adjusted when one attribute changes its value. The Notify behavior, from the Control menu, can tell other attributes watching it that they should execute their behavior, while the Watch behavior, from the Misc menu, allows one attribute to indicate that it observes another attribute.

The exact effects of all predefined behaviors are described in *Predefined Accessors*.

Alternatively, you can access this page via online help. Select Help from the menu bar. Select a behavior in the Control, Display, or Misc menus and a help page describing the effect of the behavior will appear in the left-hand pane of the panel.

Using the Edit Menu of the Behavior Page

From the Behavior page, you can also:

- Add intermediate or hidden attributes that will be used in intermediate states of the computation:

1. Select Edit > New Attribute to add an attribute.

   A new unnamed attribute appears.

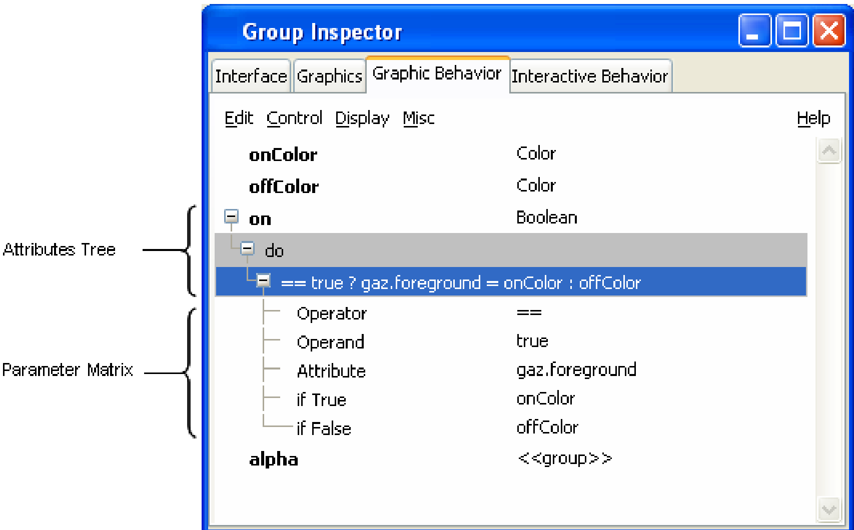
2. Set its name and type, as on the Interface page.

3. When the attribute is selected, one or more behaviors can be added to it.

- Cut/Copy/Paste behaviors. You can copy or cut a whole attribute and its behaviors by selecting the first line of an attributes tree and selecting Edit > Copy or Edit > Cut, or copy or cut a single behavior only by selecting the behavior's line and then Edit > Copy or Edit > Cut. You can paste the content of the attribute's clipboard by first selecting a line where you want the attribute to be inserted, and then selecting Edit > Paste.
- Delete selected attributes, behaviors, or parameters with Edit > Delete.
- Move behaviors up or down. Behaviors are triggered from the top to the bottom. You can decide in which order they are to be triggered.

Setting Accessor Parameters

Depending on its class, a behavior may require additional parameters to be fully defined. Behavior parameters are edited in the matrix to the right of the Attributes Tree:



The Group Inspector is designed so that you can define complex behaviors for your prototypes by simply selecting parameter values in combo boxes or dialog boxes.

Each matrix row corresponds to a parameter:

- The left column contains the parameter label.
- The right column contains the parameter values.

When a behavior is added, a parameter matrix is initialized with default or empty values that may need to be filled with appropriate values.

To edit a parameter value, click twice on the corresponding item in the matrix. This creates an editing field on top of the value item, which is either a combo box or a text field (see Figure 2.4). The combo box is initialized with relevant values for this parameter.

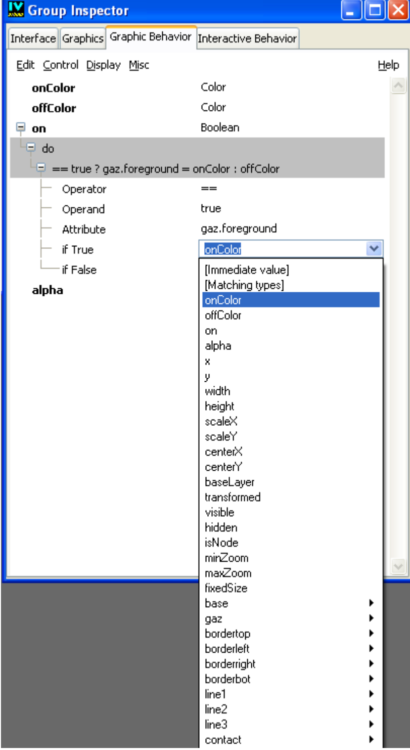
Figure 2.4 *A Combo Box With an Example of a Default Value*

There are four types of parameter, as shown in the following table:

Table 2.2 *Behavior Parameters*

Parameter Type	Description	Symbol
Literal/Explicit	The value is a string or an enumerated type that must be specified explicitly.	(e)
Input	The value is queried when the accessor is evaluated. These values can be a constant (a string or a number), a reference to other attributes, or an expression that is a combination of constants and references.	(
Output	The value is changed when the behavior is evaluated. (A call to the `changeValue` method is made.) Hence, the value must be a name that references either an existing attribute of the prototype.	)
Object/Node	The value of the parameter must be the name of an existing node. Some accessors accept only certain kinds of objects as a parameter. For instance, display behaviors act only on graphic nodes.	▯

For input parameters, the editing field is a combo box that contains a tree of accessors, as well as two special items at the beginning of the tree:



- `[Immediate value]` - When this item is selected, the editing field is set up to edit an immediate parameter value. If the value type can be determined, a value selector (that is, either a combo box or a resource selector) is created. You can also type the immediate value directly. If the value is not a number or a Boolean value, the value can be in double quotes (for example, you must enter a color as `"red"`). The value can also be an expression.

- `[All types / Matching types]` - This item toggles between the two values. `[All types]` lists all the accessors, even those whose type does not match the expected type. `[Matching types]` lists only the accessors whose type matches the expected type for the value. It is generally better to edit parameter values from top to bottom, because the editing field is often initialized using information available from the preceding fields.

Input parameters expressions can contain:

- Constants: numeric or strings literals (to be placed between quotation marks)

- Variables: prototype values or node attributes

- Arithmetic operators and parentheses: (+, -, *, **, /, %, ==, !=, >, >=, <, <=, &&, ||)

◆ Predefined functions: abs, acos, asin, atan, ceil, cos, exp, floor, log, rand, rint, round, sin, sqrt, tan. (See your standard C/C++ library user's manual to get the meaning of these functions.)

> ***Note:*** *Unlike its C / C++ equivalent, rand takes an integer argument. A non-zero argument is used by the random number generator as a seed when producing a random number. Otherwise, rand(0) returns the next integer in the random sequence started the last time the random generator was initialized.*

Notifying behaviors have the side effect of propagating the value of the attributes to their watching attributes instead of simply setting it. In this case, the behaviors of the watching attributes are evaluated in sequence. Such behaviors (Trigger accessors) show an outward arrow on top of their Output parameter, and the values they are connected to show an inward arrow:

Some behaviors can have a variable number of parameters. These accessors are identified by their last row of the parameters column, which indicates “<<Click to add item>>”. To create a new row in the editing matrix of the Group Inspector for these behaviors, press the Enter key in the value field of the last parameter or simply click in the indicated field.

Defining Interactive Behaviors

Add user behaviors using the Interaction page of the Group Inspector, to determine how user actions affect the attributes of your prototype:

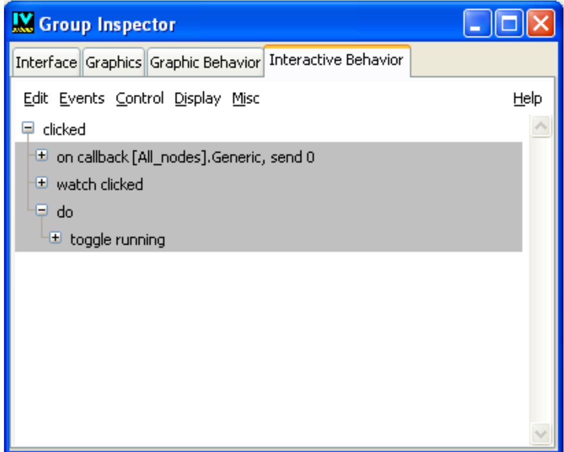


Figure 2.5 The Interaction Page of the Group Inspector.

This page works the same way as the Behaviors page and displays a list of behaviors. The page starts blank, as only the behaviors that are triggered by user actions are displayed in this page.

For each attribute, you can add behaviors, which are instructions to be performed each time the value is modified:

1. Choose an item in the Events menu to add an interactive behavior:
   - If the user action comes from a callback, select Events > Callback to add a behavior that will be triggered by some object interactor and callback.
   - Otherwise, to directly handle user events such as button clicks, select Events > Event to add a behavior that will be triggered by simple user events.
2. Enter the parameters for each added behavior.
3. Once you have added the triggering accessor, you can add behaviors that will be triggered by the user action by adding Control > item or Display > item behaviors, just as you did for the graphic behaviors.

It is generally a good idea to have the interaction accessors modify only the public attributes of your prototype, relying on that modification to update all the display behaviors.

Editing a Prototype

Once you have created and saved a prototype, you can edit it again by selecting the prototype in the palette, and choosing View > Edit Group (Ctrl+E). All the changes you make will be propagated to the instances you have created when you save it.

For instance, you can select the “bulb” prototype in the “samples” palette, and double click on its icon: the Prototype edition buffer is opened, and the group inspector allows you to edit its interface and behaviors.

Testing Your Prototype

Once you have defined behaviors for a prototype, you can test them by changing the prototype attributes:

1. Select the Interface page in the Group Inspector.
2. To set an attribute value, click the matrix item.
3. Depending on the type of value, a combo box, a resource selector, or a simple text field is created. Clicking with the right mouse button displays the list or the selector (if any) in a single click.
4. The prototype value is changed through a call to the `changeValue` method when you press the Enter key after editing the field, or when you move to another field after editing the value.

This allows you to test how the graphic representation of your prototype changes as you set its attributes to different values.

To test the interactive behavior of your prototype:

1. Switch to the active mode, as is possible for all panels.
2. By clicking and dragging on various items of your prototypes, you can see in the Group Inspector panel how the attributes are affected by the interactions you define.

Saving a Prototype

To add a prototype to an existing library:

1. Choose the Save As command from the File menu.
2. If the prototype was not already part of a library, Prototypes Studio asks if you want to add your prototype to a prototype library. Click Yes.
3. From the dialog box displayed, select the library to which you want to add the prototype.
4. In the subsequent dialog box enter the name of the prototype.

To save your prototype elsewhere than in a prototype library:

1. Choose the Save As command from the File menu.
2. Answer No to the question ‘Save the prototype in a library?’

3. A file selector dialog box appears and asks for a file name. Specify a name with a `.ivp` extension.

> ***Note:*** *A prototype which is not contained in a prototype library does not appear in the Prototypes Palette, so you cannot create instances of this prototype in panels from within IBM ILOG Views Studio. It is strongly suggested that you use prototype libraries instead of saving each prototype in its own file.*

If you make additional changes to your prototype, you can save it again in the same prototype library or to the same file using the Save command from the File menu.

You can move the prototype to a different library, give it a different name, or remove it from its library with the Save As command.

When a prototype is saved, all the panels containing instances of that prototype are updated with the new prototype definition.

Loading and Saving Prototype Libraries

To load a prototype library:

1. Choose Open from the File menu.
2. Select Prototype libraries (`*.ipl`) in the box for file type.
3. Browse to find the name of the library file you want to load.

   Once the library is loaded, it is added as a new palette in the Prototypes palette of the Palettes panel.

You do not need to save a prototype library each time you create or edit a prototype. A prototype library is saved automatically as necessary while you edit your prototypes.

To change the name of the current prototype library (that is, the library name that is displayed in the visible page of the Prototypes Palette):

1. Use the command Save Prototype Library As... from the File menu.
2. Select the new directory and name of the prototype library file (with a `.ipl` extension).

   The prototype library name changes accordingly and all the prototypes of the library are saved in the new directory.

You can close a prototype library that you no longer need by choosing Close Prototype Library from the File menu. This command removes the library currently displayed in the Prototypes palette.

> ***Note:*** *The prototypes contained in the library are not actually deleted in memory; they can still be referenced in panels or in other prototypes.*

Creating and Editing Prototype Instances in Panels

This section explains how to instantiate the prototypes that you have defined or loaded, in order to create panels.

Choosing a Buffer Type

Prototypes Studio has 2 types of buffer windows that can be made into panels: 2D Graphics and Grapher. When the Gadgets extension is installed, a Gadget buffer window is also available.

- Use a 2D Graphics buffer window for graphics-intensive applications: that is, if your prototypes contain 2D graphic objects such as lines, rectangles, and splines.
- Use a Grapher buffer window if you need Grapher features to connect graphic objects in prototype instances, using Grapher links.
- If you have the IBM ILOG Views Controls package and your prototypes contain gadgets, use a Gadget buffer window.

To create a panel in which to use prototype instances, choose the appropriate buffer type from the menu File > New.

Creating a Prototype Instance

To create a prototype instance:

1. Select a prototype library in the upper pane of the Palettes panel.
2. Drag the icon of the desired prototype to the buffer window.

OR:

1. Click the icon representing a prototype to select it.
2. Click and drag a rectangle in a buffer window. An instance of the prototype whose bounding box is defined by the rectangle you just drew will be created.

The prototype is instantiated in the form of an `IlvProtoGraphic` object encapsulating the prototype instance.

Editing Prototype Instances

Prototype instances are edited using the Group Inspector. To display the Group Inspector, choose Group Inspector from the Tools menu or double-click a prototype instance. When an instance is selected, its attributes are displayed in the Attributes notebook page of the Group Inspector.

> ***Note:*** *The Behavior and Interaction pages are disabled for prototype instances. They can only be used when editing prototypes. See Defining Graphic Behaviors for an explanation on how to edit accessor values with the Group Inspector.*

Loading and Saving Panels

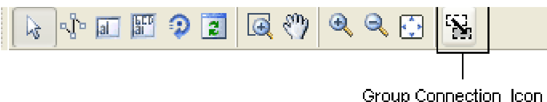
Panels containing prototype instances are loaded and saved as standard `.ilv` files using the Open, Save, and Save As commands from the File menu.

Connecting Prototype Instances

Prototypes can define notifying attributes that can be connected to the attributes of instances of other prototypes. This means that when a notifying attribute is modified, its value is assigned to the attributes of the objects it is connected to.

To connect attributes of prototype instances:

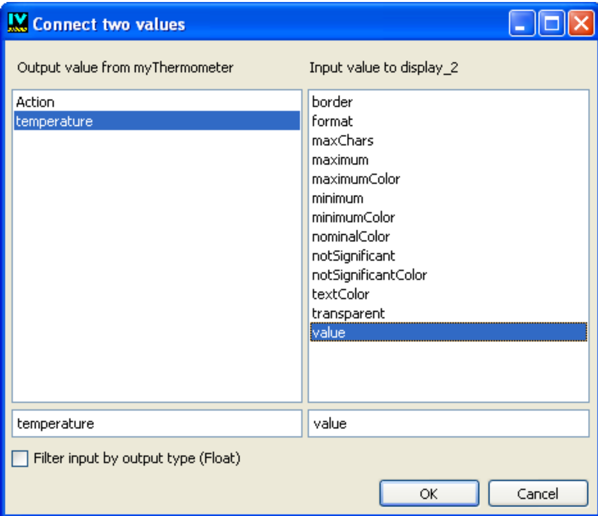
1. Select the Group Connection mode from the Editing Modes toolbar:



2. Click in the prototype instance that defines the notifying attribute (the value that you want to be sent).

3. Drag the connection line to the instance to which you want to connect this attribute (the instance that you want to be notified of attribute modifications).

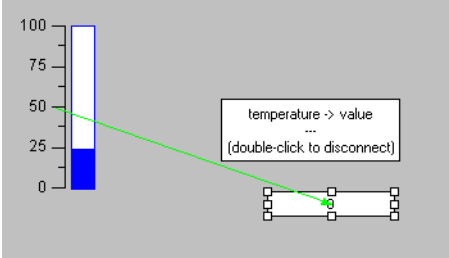
   The Connect two values dialog box is displayed:



4. In the left hand pane, select the notifying attribute from the first instance.

5. In the right hand pane, select the input attribute for the second instance.

There can be several connections between the same two objects. When the Group Connection mode is active, existing connections are displayed as green lines. If you click a green line, the connection details (that is, the names of the output and input values) are displayed.



To delete a connection:

1. Double-click the connection line.

A Delete Connection dialog box appears.

2. Select the connection you want to delete, and click Apply.

The next chapter will describe how to link protoypes to application objects, defined in C++ code.

C H A P T E R

3

The User Interface and Commands

This chapter introduces you to the Prototypes extension of IBM® ILOG® Views Studio, an extension designed to facilitate the development of fully interactive graphical user interfaces of domain-specific application objects.

You can find information on the following topics:

- *Overview*
- *The Main Window*
- *The Palettes Panel*
- *Group Inspector Panel*
- *Prototypes Extension Commands*

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

Overview

The Prototypes extension of IBM® ILOG® Views Studio allows you to define complex graphic objects, called prototypes, by interactively assembling IBM® ILOG® Views

graphic objects. Behaviors can be attached to these prototypes to define the whole graphical and interactive part of your application.

The prototypes can be instantiated and used as basic building blocks for application windows, object inspectors, or direct-manipulation interfaces, that is, when each application object is directly tied to an interactive graphical representation.

IBM ILOG Views Studio with the Prototypes extension defines a new workflow to build highly interactive user interfaces: you develop the interactive part of your application in a graphical editor, store it in libraries, and then link it with your core functionality written in C++.

Launching IBM ILOG Views Studio With the Prototypes Extension

If you have installed the 2D Graphics Pro package, the Prototypes extension is automatically loaded when IBM ILOG Views Studio is launched. The extension is called `smproto` in the configuration file. A compatibility extension enabling now deprecated features can also be used: `smoldpro`. This manual describes only the features of the `smproto` extension. To install or uninstall these extensions, see the section on Loading Plug-Ins in *Introducing IBM ILOG Views Studio*.

The Main Window

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

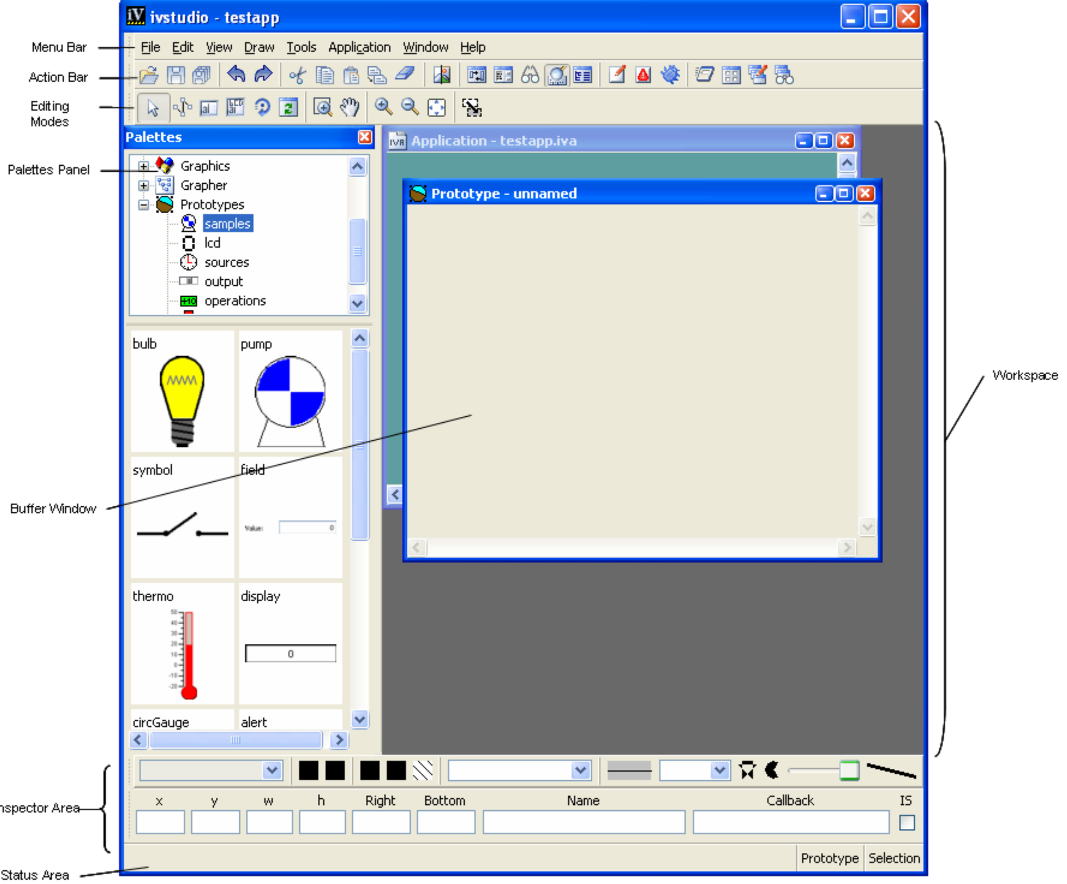


Figure 3.1 IBM ILOG Views Studio Main Window with Prototypes Extension at Start-up

The Main window appears much as it does when only the IBM ILOG Views Studio Foundation package is installed. However, you will notice that with the Protoypes 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. These are now briefly presented.

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 Prototypes extension of IBM ILOG Views Studio, you can edit the following types of buffers:

- 2D Graphics
- Grapher
- Prototypes

An empty 2D Graphics buffer is displayed by default when you launch IBM ILOG Views Studio.

As you switch between the buffers currently loaded in the Main window, you will notice that 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 2D Graphics Buffer Window

The 2D Graphics buffer is the default for the Foundation package. 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 Grapher Buffer Window

The Grapher buffer window lets you edit the contents of an `IlvGrapher`. It uses an `IlvGrapher` to load, edit, and save nodes and links.

To create a new Grapher buffer window:

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

To open this window, you can also execute the `NewGrapherBuffer` 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 `IlvGrapher`, a Grapher buffer window is automatically opened.

The Prototypes Buffer Window

The Prototypes buffer window is used to create and manipulate your prototypes. Graphic objects, and Prototypes, can be dragged from a Palettes panel to an active Prototypes buffer window.

To open a new Prototypes buffer window:

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

Alternatively, when you double-click a Prototype in a Prototypes palette, a Prototypes buffer window is automatically opened to allow you to modify it or inspect its attributes and behaviors.

The Menu Bar

When the Prototypes package is installed, additional commands are available through the menu bar in the Main window.

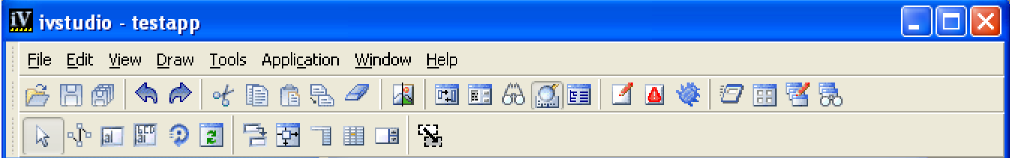


Figure 3.2 *IBM ILOG Views Studio Prototypes Extension Menu Bar*

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

File Menu Commands

Menu Item	Command
New > Prototype	NewPrototypeEditionBuffer
New > Prototype Library...	NewProtoLibrary
New > Prototype Grapher	NewPrototypeGrapherBuffer *Note: This is a deprecated command and is provided purely for compatibility with earlier versions.*
Save Prototype Library as...	SaveProtoLibraryAs
Close Prototype Library	CloseProtoLibrary

Draw Menu Commands

Menu Item	Command
Group	GroupIntoGroup
Edit Prototype	EditPrototype
Delete Prototype	DeletePrototype

Tools Menu Command

Menu Item	Description
Group Inspector	This opens the Group Inspector of the currently selected prototype instance or `IlvGroup` object.

View Menu Command

Menu Item	Command
Toggle Animation Timers	ToggleTimers

The Action Toolbar

The Action toolbar remains unchanged from the Foundation package.

The Editing Modes Toolbar

The Prototypes extension of IBM ILOG Views Studio contains an editing mode in addition to the regular IBM ILOG Views Studio editing modes:

Group Connection Mode

Use the Group Connection mode for connecting the values of prototype instances. The Connection mode is used to define connections between prototype instances.See *Connecting Prototype Instances*.

The Palettes Panel

The Prototypes palette is included in the Palettes panel, as shown in Figure 3.3. It shows the various prototype libraries that you have defined or loaded, and allows you to instantiate prototypes by dragging their icons to Prototype buffers. Prototypes are manipulated like other graphic objects. Each library defines its own panel in the Palettes panel.

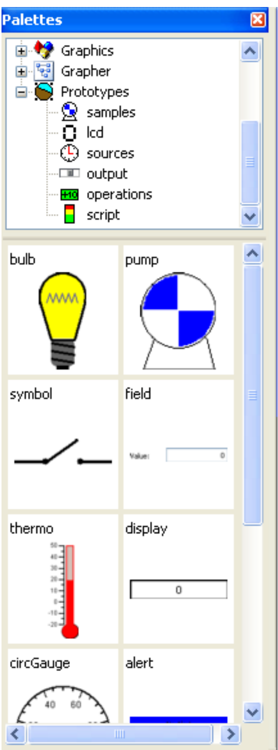


Figure 3.3 The Palettes Panel Showing the `samples` *Prototype Library*

When the Prototypes extension is installed, IBM ILOG Views Studio loads the following libraries at start-up:

Library	Description
`samples`	Sample library loaded at start-up.
`sources`	Prototypes containing value sources.
`output`	Prototypes containing gadgets and defining output values.
`lcd`	LCD displays (one digit and four digits).
`operations`	Prototypes that can be used to connect prototypes and execute operations on their values.
`script`	Prototypes that use script accessors.

To open one of these prototype libraries, go to the upper pane of the Palettes panel and click the name in the Prototypes palette.

You can look at any prototype definition by double-clicking the prototype icon. This will load the prototype into a Prototype buffer window and open a Group Inspector panel.

> ***Note:*** *When you load a panel file that contains prototype instances, the required prototype libraries are automatically loaded in the Prototypes palette.*

The `<ILVHOME>/samples/protos` directory provides other samples of how to use prototypes.

Group Inspector Panel

The Prototypes extension provides an additional panel to let you define the interface and the graphic and interactive behaviors of your prototypes, as shown in Figure 3.4. It can also be used to customize groups and prototype instances.

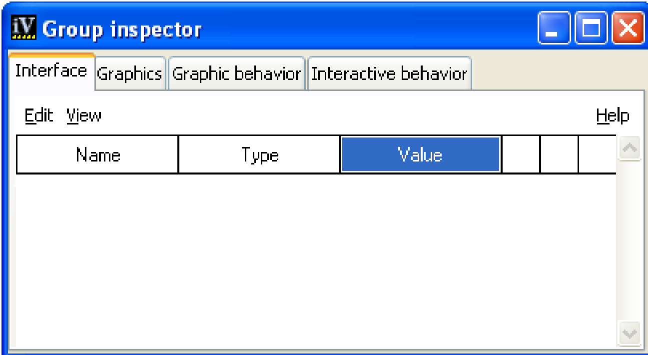


Figure 3.4 *The Group Inspector Panel*

Access to Panel

The panel is accessed by one of the following methods:

- Choosing Group Inspector from the Tools menu.
- Creating a new Prototype buffer window.
- Double-clicking a prototype in a Prototypes buffer window.
- Choosing Commands from the Tools menu, selecting the `ShowGroupInspector` command in the list, and clicking Apply.

Group Inspector Elements

The Group Inspector panel has four notebook pages:

- The Attributes page is used to define the attributes of a prototype and to customize prototype instances.
- The Graphics page is used to display and edit the graphic objects composing a prototype.
- The Behavior page is used to define the graphic behavior of a prototype.
- The Interaction page is used to define the interactive behaviors of a prototype.

Full context-sensitive hypertext help is available when you click Help on the inspector. This help page can be hidden by clicking the Close Help button.

The features of the Group Inspector panel are detailed in *Creating and Using Prototypes*.

Prototypes Extension Commands

This section presents an alphabetical listing of the additional, predefined commands that are available in the Prototypes 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, how to access it if it is accessible other than through the Commands panel, the category to which it belongs, 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.

CloseProtoLibrary

Label	Close Prototype Library
Path	Main window: File menu
Category	prototypes
Action	Closes the prototype library currently displayed in the Palettes panel.

ConvertProtoManager

Label	Convert ProtoManager
Path	Convert ProtoManager: Edit menu

Category	prototypes
Action	Creates a new regular Studio buffer that copies the content of the currently active Prototype instance buffer. This command uses the `IlvPrConvertProtoManager` function to perform the conversion. It is meant to help in switching from Views 3.1 prototypes to the more recent API.

DeletePrototype

Label	Delete Prototype
Path	Main window: Edit menu
Category	prototypes
Action	Removes the selected prototype from its library.

EditPrototype

Label	Edit Prototype
Path	Main window: Edit menu
Category	prototypes
Action	Edits the selected prototype in a new Prototype buffer window, and opens the Group Inspector panel for the prototype instance.

GroupIntoGroup

Label	IlvGroup
Path	Main window: Draw menu > Group
Category	prototypes
Action	Groups the selected objects into an `IlvGroup`.

NewProtoLibrary

Label	Prototype Library...
Path	Main window: File menu > New
Category	prototypes
Action	Creates a new prototype library. This library is visible in the Palettes panel. A file selector dialog box is opened to choose the library file (`.ipl`).

NewPrototype

Label	New Prototype
Path	Main Window: File menu > New
Category	prototypes
Action	Creates a buffer window used to draw and edit prototypes.

> ***Note:*** *This is a deprecated command and is provided purely for compatibility with earlier versions.*

NewPrototypeEditionBuffer

Label	Prototype
Path	Main window: File menu > New
Category	prototypes
Action	Creates a buffer window used to draw and edit prototypes.

NewPrototypeGrapherBuffer

Label	Prototype Grapher
Path	Main window: File menu > New

Category	prototypes
Action	Allows you to create an instance of an `IlvProtoGrapher` class.

> ***Note:*** *This is a deprecated command and is provided purely for compatibility with earlier versions.*

OpenProtoLibrary

Label	Open Prototype Library...
Path	Main window: File menu > Open
Category	prototypes
Action	Opens a prototype library file. A selection dialog box is opened to choose the `.ipl` file to open.

SaveProtoLibraryAs

Label	Save Prototype Library As...
Path	Main window: File menu
Category	prototypes
Action	Saves a copy of the currently selected prototype library to a different file.

SelectGroupConnectionMode

Label	Group Connection
Path	Editing Modes toolbar
Category	prototypes
Action	Selects the Group Connection mode.

SelectGroupSelectionMode

Label	Group Selection
Path	Editing Modes toolbar
Category	prototypes
Action	Selects the Group Selection mode.

SelectNodeSelectionMode

Label	Node Selection
Path	Editing Modes toolbar
Category	prototypes
Action	Selects the Node Selection mode in a Prototype buffer. This mode lets users select and inspect graphic nodes while editing a prototype.

ShowGroupEditor

Label	Group Inspector
Path	Tools menu
Category	prototypes
Action	Shows/hides the Group Inspector panel.

ToggleTimers

Label	Toggle Animation Timers
Path	View menu

Category	prototypes
Action	Turns on or off the animation timers of the prototype's animation accessors, thereby allowing you to edit the prototype and then test its behavior.

UngroupIlvGroups

Label	Ungroup
Path	Main window: Draw menu
Category	prototypes
Action	This command replaces the generic Ungroup command to take into account `IlvGroup` objects.

CHAPTER

4

Using Prototypes in C++ Applications

This chapter explains how to use prototypes in your C++ applications. It is divided as follows:

- *Architecture*
- *Writing C++ Applications Using Prototypes*
- *Linking Prototypes to Application Objects*
- *Advanced Uses of Prototypes*

Architecture

The Prototypes package is defined on top of the IBM® ILOG® Views Foundation package and allows you to perform the following tasks:

- Assemble elementary graphic objects into groups (class `IlvGroup`).
- Specify the behavior of your groups using predefined accessor objects or scripts.
- Define prototypes and create prototype instances in managers. Prototypes are instances of a subclass of `IlvGroup` called `IlvPrototype`.
- Connect properties between prototype instances.

◆ Link application objects and prototype instances.

The architecture of the Prototypes package is shown in Figure 4.1:

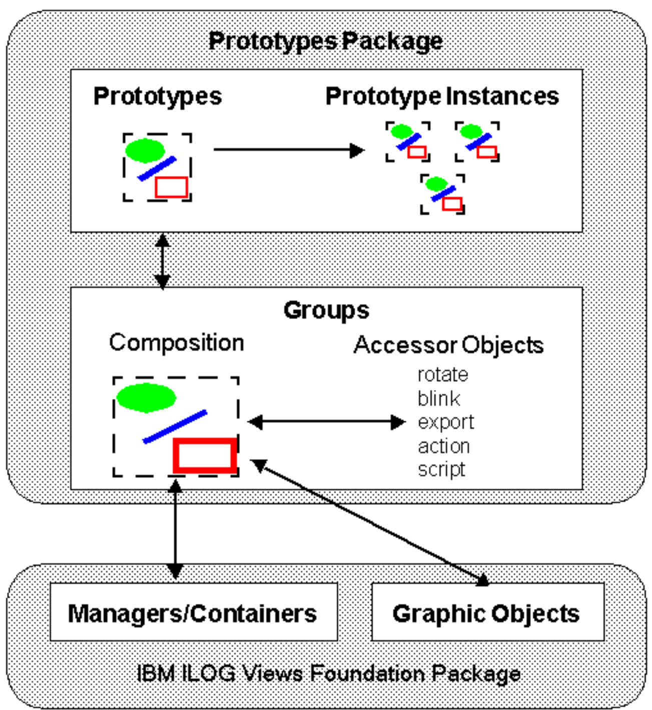


Figure 4.1 *Architecture of the Prototypes Package*

Groups

Groups are the basic components of the Prototypes package.

To create a BGO with the Prototypes package, you must first assemble basic IBM ILOG Views graphic objects to build a group. You can use any IBM ILOG Views graphic object in a group. You can also create subgroups to build structured objects.

A group is represented in C++ by the `IlvGroup` class. An `IlvGroup` object contains a hierarchy of nodes, represented by the following subclasses of the `IlvGroupNode` class:

◆ An `IlvGraphicNode` is a node that holds a graphic object (an instance of a subclass of `IlvGraphic`). A group contains one graphic node for each of its graphic elements.

- An `IlvSubGroupNode` holds a subgroup, that is, a group contained in another group. This class is used to create object hierarchies.

> ***Notes:*** `IlvGroup` *objects are different from* `IlvGraphicSet` *objects. An* `IlvGroup` *is a logical hierarchy of graphic objects that are contained in a manager. Unlike* `IlvGraphicSet`*,* `IlvGroup` *is not a subclass of* `IlvGraphic`*. An* `IlvProtoGraphic` *is a subclass of* `IlvGraphic` *intended to encapsulate an* `IlvGroup` *to place it in a manager or container.*
>
> *A third class of* `IlvGroupNode`*, called* `IlvValueSourceNode`*, is still present in the package, but its use is deprecated.*

Attributes and Accessor Objects

The Prototypes package lets you define not only the graphic appearance of your objects, but also their behavior. The behavior of a group is controlled by its attributes (also called properties). These attributes bear distinct names and represent the external interface of the group, that is, how its appearance will be controlled from your application.

The attributes of a group and their behaviors are defined in accessor objects. Each accessor object has a name and a type and implements the effect of setting and/or retrieving the value for the group. Several accessor objects can have the same name, which means they belong to the same attribute. This means that setting an attribute value can have several side effects.

Accessors can be linked to other attributes of objects or to application data. They define state or appearance changes in response to user events or application instructions and, by extension, specify the graphic and interactive behavior of objects. Accessor objects are instances of subclasses of `IlvAccessor`.

In other words, the relationship between accessor objects and values is the following:

- You interact with a group through its attributes.
- A group has a set of accessor objects attached to it. Each accessor object is associated with a name, which defines an attribute (or facet) of the BGO.
- The `IlvGroup::changeValue` method calls the `changeValue` methods of all the accessor objects of a given name, thereby *setting* the attribute value.
- The `IlvGroup::queryValue` method calls the `queryValue` methods of all the accessor objects of a given name, thereby *getting* the attribute value.
- The effect of each behavior class is defined by the implementation of its `changeValue` and `queryValue` methods.
- Some accessors can be set through user interaction or by the application, thereby triggering other behaviors in a chain.

The relationship between accessor objects and attributes/behaviors in a group is shown in Figure 4.2:

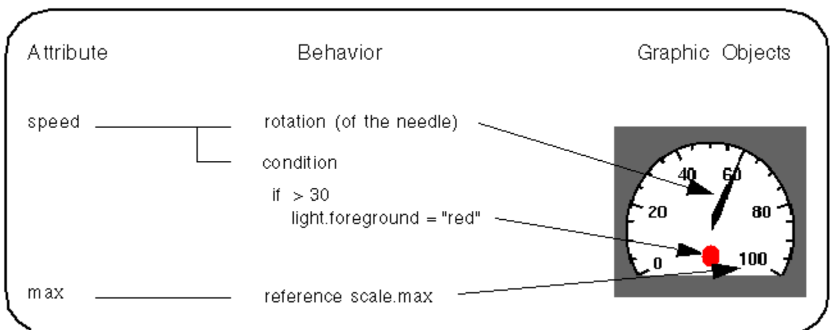


Figure 4.2 *Relationship Between Accessor Objects, Attributes, and Behaviors*

This example shows a group representing a gauge. The gauge has two attributes: `speed` and `max`.

- The `speed` attribute is implemented by two accessor objects, each having a graphic behavior:
  - A Rotation accessor object—when the `speed` attribute is changed, this accessor object rotates the needle of the gauge.
  - A Condition accessor object—when the `speed` attribute is changed, this accessor object changes the color of the circle if the value is greater than 30.
- The `max` attribute is implemented by a Reference accessor object, which references a property of a basic graphic object at the group level. When the `max` attribute is changed, this accessor object changes the maximum speed of the scale graphic object.

The following types of accessors can be attached to an attribute:

- **Data accessors** - State how a data item is to be stored (locally or in a node) and what its type is. They are comparable to variable declarations in a regular programming language. Only one of these accessors should be present for each attribute.
- **Control accessors** - Perform conditional instructions, evaluations, and assignments based on other attributes. They take input parameters and can have output effects on other parameters. Typical examples are the Condition accessor, for the conditional assignment, and the Toggle accessor.
- **Notifying accessors** - Define the entry points of evaluation cycles. Either the application (when it does a `pushValue`) or the user (when a callback triggers a Callback accessor) can “push” values, forcing the accessors to handle them. Connections between attributes can be made to propagate the evaluation to other values by means of the Watch and Notify accessors.

- **Display accessors** - Define a side effect of the attribute on the visual presentation of the nodes of the group. They correspond to calls to the drawing library in a programming language. When they are set, they change graphic properties, such as rotating a node or changing the visibility of a graphic component.
- **Animation accessors** - A special case of display accessors that periodically change a graphic attribute.
- **Miscellaneous accessors** - Consist of two accessors that do not fit into the previous categories: the Debug accessor and the Delegate to Prototype accessor.

You will find a full description of all the predefined accessor classes in the section *Predefined Accessors*.

Accessor Parameters

Accessors define a side effect that is performed on another object or attribute when a given attribute is set. This means that, as with a function in a programming language, an accessor has to take parameters to customize its effects. A description of the four types of parameters that accessors can have is presented in Table 4.1.

Table 4.1 *Accessor Parameters*

Parameter Type	Description
Direct parameters	The value is a string or an enumerated type that must be specified explicitly.
Input parameters	The value is queried when the accessor is evaluated. These values can be a constant (a string or a number), a reference to other values (attributes of nodes or prototype values), or an expression that is a combination of constants and references.
Output parameters	The value is changed when the accessor is evaluated (A call to the `changeValue` method is made). Hence, the value must be a name that references either an existing attribute of a node or a prototype value.
Object/Node parameters	The value of the parameter must be the name of an existing node. Some accessors accept only certain kinds of objects as a parameter. For instance, display accessors act only on graphic nodes.

Input parameter expressions can contain:

- **Constants**: numeric or string literals
- **Variables**: prototype values or node attributes
- **Arithmetic operators and parentheses**: (+, -, *, **, /, %, ==, !=, >, >=, <, <=, &&, ||)
- **Predefined functions**: abs, acos, asin, atan, ceil, cos, exp, floor, log, rand, rint, round, sin, sqrt, tan

> ***Note:*** *Contrary to its C/C++ equivalent, rand() takes an integer argument. If this argument is non-zero, it is used as a seed for the random generator before a random number is generated. Otherwise, rand(0) returns the next integer in the random sequence started the last time the random generator was initialized.*

Prototypes and Instances

Once you have defined the graphic contents and the behavior of a group, you can save it as a prototype. A prototype is the model of a BGO. Usually, you create prototypes with IBM ILOG Views Studio, although you can also create prototypes directly by coding. See *Creating Prototypes by Coding*.

Prototypes are stored, loaded, and saved using prototype libraries, represented by the class `IlvProtoLibrary`. You can create prototype instances from a prototype. A prototype instance is a full copy of its prototype.

Prototypes are represented by the `IlvPrototype` class and prototype instances are represented by the `IlvProtoInstance` class. Both of these classes are subclasses of `IlvGroup`.

When a prototype instance is saved to a file, the manager writes only the values of the properties that have been modified for that instance. The graphic objects that compose the prototype instance are not saved to the file. This means that you can completely change the definition of the prototype, add or remove graphic objects, and so on. Instances of the modified prototype will be automatically updated with the new definition.

Displaying Groups and Instances in Managers and Containers

To display groups, prototypes, and instances in a panel of your application (an `IlvManager` or `IlvContainer`), you need to place them in an `IlvProtoGraphic` object and add this object to the manager or container. `IlvProtoGraphic` is a subclass of `IlvGraphic` designed to encapsulate all the graphic objects of a group. You may also add group objects to a manager using an `IlvGroupHolder`, which is a class that extends the properties of a container or manager and lets you directly add or retrieve groups through convenience

functions.This class will create the `IlvProtoGraphic` itself to wrap it around the `IlvGroup`.

> ***Note:*** *Special manager and container classes* `IlvProtoManager`*,* `IlvProtoContainer`*, and* `IlvProtoGrapher` *have been added to allow direct handling of* `IlvGroup` *objects in a container or manager. These classes are provided for compatibility reasons. Their use is obsolete and should be avoided.*

Connecting Attributes

A group has readable and writable attributes that are defined by the accessor objects attached to it. It can also have notifying attributes, which are similar to events generated by the group or by one of its elements.

A notifying attribute can be connected to an attribute of another group. When the attribute is modified, the changes are propagated to the groups connected to it. This is referred to as the value of the notifying attribute being pushed to its connected attributes.

Linking Application Objects to Prototypes

Once you have defined your prototypes and your panels, you may want to connect these to real application data and processes defined in C++.

There are three methods available to link prototypes to application objects, depending on the type of interface you want to produce:

- When the display is graphics-rich but represents only a few application objects and values, you may want to link the application objects by directly feeding values to the prototype instances of a given panel.

  This is typically used in static synoptic displays composed of only predefined graphic components. It is convenient to use feed values directly when the application is not expecting user input to modify application values through a prototype instance. The `base_feed` sample in `<ILVHOME>/samples/protos` shows how to use this approach for a simple control panel.

- To build WYSIWYG, direct-manipulation application object editors, you may want to use an `IlvGroupMediator`. With this class, you can link an application object to a given `IlvGroup` (or prototype instance) in a panel, allowing interactive editing of its attributes. A group mediator allows you to bind and unbind application objects dynamically to a given prototype that serves as an editor for the object.

  A typical application of this type is a WYSIWYG inspector such as the Guides inspector in Prototypes Studio. The `inspector` sample in `<ILVHOME>/samples/protos` is an example of this kind of editor. It shows how to build a 2D transformation matrix editor controlling the viewpoint of a view interactively.

- To create many instances of a prototype dynamically, with each instance linked to a given application object, you should use an `IlvProtoMediator`. This class instantiates the prototype and links it to an application object of a given class as it is created. This allows clear separation of interface design from the application design, each being able to evolve separately from a commonly agreed upon application interface.

  A typical application of this type allows you to view panels where many objects of many classes are represented and edited at the same time. Each application class is linked to a prototype and each instance of the class to an instance of the prototype.

  Cartographic displays and all graph displays are examples of applications that can benefit from prototypes using `IlvProtoMediators`. `<ILVHOME>/samples/protos/interact_synoptic` is an example of this type of application, showing a very simple air-traffic simulator, where each flight and each airport are represented by prototype instances. The simulator only deals with changing the attributes of the flight, whereas the prototypes can be incrementally refined in the drawing editor to present the best display.

Writing C++ Applications Using Prototypes

As a general rule, you create your prototypes in IBM® ILOG® Views Studio and then use them in your application. The following section explains the C++ API that you use to add prototypes to your application and how to manipulate these prototypes.

> ***Note:*** *Although it is not the general rule, it is possible to create prototypes through direct coding. For these situations, see the section Creating Prototypes by Coding.*

The following items are described in this section:

- *Header Files*
- *Loading a Panel Containing Prototype Instances*
- *Loading Prototypes*
- *Creating Prototype Instances*
- *Deleting Prototype Instances*
- *Retrieving Groups and Prototype Instances*
- *Getting and Setting Attributes*
- *User-Defined and Predefined Attributes*

Header Files

To make sure that your application is linked with the necessary library packages, you must first include the header files corresponding to the graphic objects, the accessors (subclasses of `IlvUserAccessor`), and the interactors used in the prototypes that you will load.

To include all the predefined accessor classes, use the header file `<ILVHOME>/include/ilviews/protos/allaccs.h`.

Here is a typical set of header files to include in order to build an application that can load prototypes containing any type of graphic object:

```
#include <ilviews/protos/protogr.h> // for IlvProtoGraphic.
#include <ilviews/protos/allaccs.h>  // for all accessors.
#include <ilviews/graphics/all.h> // for all graphic objects.
#include <ilviews/gadgets/gadgets.h> // if you use gadgets in your prototypes.
#include <ilviews/graphics/inter.h> // for all object interactors.
```

You may also want to add the following header files:

```
#include <ilviews/protos/groupholder.h> // to get the groups attached
                                        // to a given container or manager.

#include <ilviews/protos/proto.h> // to manipulate prototypes and
                                  // their libraries.

#include <ilviews/protos/grouplin.h> // to attach prototypes to
                                     // application objects.
```

If you know in advance the Prototypes that you will use, you can reduce the size of your application by including only the necessary header files instead of `allaccs.h`, `graphics.h`, and `gadgets.h`.

To compile applications that use the prototypes package, you must compile them with the library `ilvproto`. This library also requires the following libraries: `ilvgrapher`, `ilvmgr`, and the usual IBM ILOG Views libraries for your platform. The `ilvgdpro` library may be needed for applications that use old features of prototypes.

Loading a Panel Containing Prototype Instances

To load a `.ilv` file containing prototype instances, you simply use the `read` or `readFile` methods of `IlvManager` or `IlvContainer`:

```
Container->readFile("protoSample.ilv");
```

All the prototypes used in the file will be loaded automatically from their prototype libraries. Prototype libraries are searched for in the file system using the display path. For example, if the panel contains prototypes from a prototype library called `mylib` located in `/usr/home/yourdir/protolibs/mylib.ipl`, you should include `/usr/home/yourdir/protolibs/` in your `ILVPATH` environment variable.

To allow handling of groups in a container or manager, the `IlvGroupHolder` class provides all the necessary interface. Instances of the `IlvGroupHolder` class are automatically attached to containers or managers containing prototype instances. This class provides the methods for adding, removing, and retrieving groups (and thus prototype instances). You can retrieve the group holder attached to a container, manager, or graphic holder with the global methods:

- `IlvGroupHolder* groupHolder = IlvGroupHolder::Get(manager);`
- `IlvGroupHolder* groupHolder2 = IlvGroupHolder::Get(manager ->getHolder());`
- `IlvGroupHolder* groupHolder3 = IlvGroupHolder::Get(container);`

Loading Prototypes

You may need to create instances of your prototypes by coding. To create instances of your prototypes, you must first load them. You can load a whole prototype library and then load one or more of the prototypes it contains. To do this, create an instance of the `IlvProtoLibrary` class and call its `load` method:

```
IlvProtoLibrary* lib = new IlvProtoLibrary(display, "mylib");
if(!lib->load())
    IlvFatalError("Could not load prototype library");
```

If you want to load a prototype library that is not located in the display path, you can specify the directory where the library is located in the call to the constructor:

```
IlvProtoLibrary* lib = new IlvProtoLibrary(display, "mylib",
                                           "/usr/somewhere/protos");
if(!lib->load())
    IlvFatalError("Could not load prototype library");
```

Once you have loaded a prototype library, you can retrieve all its prototypes or a particular prototype with the following methods:

```
IlUInt count;
IlvPrototype** protos = lib->getPrototypes(count);
```

or:

```
IlvPrototype* proto = lib->getPrototype("myproto");
```

The array returned by the `getPrototypes` method is allocated with the `new[]` operator and must be released with the `delete[]` operator when it is no longer needed.

Alternatively, you can load each prototype individually with the global function `IlvLoadPrototype`:

```
IlvPrototype* proto = IlvLoadPrototype("mylib.myproto", display);
```

The first argument specifies the name of the prototype library and the name of the prototype (separated by a period). The second argument is the instance of `IlvDisplay` that your application has created. The prototype library file and the prototype files are searched for in the file system using the display path.

Creating Prototype Instances

To create an instance of a prototype, use the `clone` method:

```
IlvPrototypeInstance* instance = proto->clone("myinstance");
```

The argument of the `clone` method is the name of the new instance. You can pass `0`, which means that a name is generated automatically.

Instances of the `IlvGroupHolder` class are automatically attached to containers or managers containing prototype instances. This class provides the methods for adding, removing, and retrieving groups (and thus prototype instances).

To add the new prototype instance to a manager or a container, you can use the `addGroup` methods of the `IlvGroupHolder` attached to manager/container classes:

```
IlvGroupHolder* groupHolder = IlvGroupHolder::Get(manager);
groupHolder->addGroup(instance);
```

Alternatively, you can create an `IlvProtoGraphic` object and directly place it in the manager.

```
IlvPrototype* proto;
// Create an instance of the prototype proto and places it
IlvProtoGraphic* protoGraphic1 = new IlvProtoGraphic(proto);
// Create an instance of a prototype
IlvProtoInstance* protoInstance = proto->clone(“instance2”);
IlvProtoGraphic* protoGraphic2 = new IlvProtoGraphic(protoInstance);
manager->addObject(protoGraphic1);
manager->addObject(protoGraphic2);
```

Often, you will set the position of the prototype instance when you add it to a manager or container. You can do this by either:

- Moving the `IlvProtoGraphic`:

  ```
  manager->moveObject(protoGraphic1, 100, 100).
  ```

- Setting the x and y attributes of the prototype instance. See *Getting and Setting Attributes* for an explanation on how to set several values in a single call.

Deleting Prototype Instances

To remove a prototype instance from its container or its manager, you can use the `removeGroup` methods of the `IlvGroupHolder` class:

```
groupHolder->removeGroup(instance);
```

You can also remove its embedding IlvProtoGraphic from its container or manager:

```
manager->removeObject(protoGraphic);
```

To completely delete a prototype instance, simply call the delete operator. You can also delete its encapsulating protoGraphic.

Retrieving Groups and Prototype Instances

To get all the groups contained in a manager or container, use the getGroups method of its attached group holder:

```
IlUInt count;
IlvGroup** instances = groupHolder->getGroups(count);
```

> ***Note:** The array of pointers returned by the* getGroups *method is allocated using the* new[] *operator and must be deleted with the* delete[] *operator when it is no longer needed.*

To retrieve a group by its name, use the getGroup method:

```
IlvProtoInstance* pump = (IlvProtoInstance*)groupHolder->getGroup("pump");
```

This method returns 0 if the specified group does not exist.

Getting and Setting Attributes

Prototype instances are manipulated through a uniform API based on named attributes (also called properties or accessors). This API is the same as the one provided by the class IlvGraphic and basically consists of the IlvGraphic::changeValue and IlvGraphic::queryValue methods.

A named attribute is represented by an instance of the IlvValue class and is defined by the following:

- The attribute name, “label” for example, to access the label of a button.
- A value, which can be of different types (for example, a character string, an integer, or a pointer).
- A type that corresponds to the type of the data.

The type of the value is set automatically by the IlvValue class. You use constructors to initialize values of predefined types (such as IlInt, const char*, IlvColor*, and so on). You can also change a value using the assignment operator = or by casting an IlvValue to a predefined type. The IlvValue class handles all conversions automatically. For more details, see the IlvValue class.

To set a value for a prototype instance, you must create an `IlvValue` and call the `IlvGraphic::changeValue` method:

```
IlvValue xval("x", (IlInt)100);
instance->changeValue(xval);
```

> ***Note:** The explicit cast of the value* `100` *to the type* `IlInt` *is necessary because an ambiguity exists between the integer and Boolean types. Without the cast, the compiler might (on some platforms) call the constructor that creates an* `IlvValue` *of type* `IlBoolean`*. It is recommended that you always use explicit casts when using constants to initialize an* `IlvValue`*.*

You do not need to create a new `IlvValue` every time you want to change a value. You can use an existing `IlvValue` and change its data with the assignment operator:

```
xval = (IlInt)200;
instance->changeValue(xval);
```

You can set several values in a single call. To do this, you must create and initialize an array of `IlvValue` objects and call the `changeValues` method. The following example shows how to set the position of an object in a single call:

```
IlvValue vals[] = {
                IlvValue("x", (IlInt)100),
                IlvValue("y", (IlInt)200)
};
instance->changeValues(vals, 2);
```

To retrieve a value, use the `queryValue` method:

```
IlvValue xval("x");
IlInt x = instance->queryValue(xval);
```

The `queryValue` method takes an `IlvValue` reference as parameter. The `IlvValue` must be initialized with the name of the value to retrieve. The `queryValue` method stores the retrieved value in its argument and returns a reference to it. In the example, assigning the result of `queryValue` to the integer variable `x` calls the `IlvValue` to the `IlInt` cast operator.

To retrieve several values in a single call, create an array of `IlvValue` objects and call the `queryValues` method:

```
IlvValue vals[] = { "x", "y", "width", "height" };
instance->queryValues(vals, 4);
IlInt x = vals[0];
IlInt y = vals[1];
IlUInt width = vals[2];
IlUInt height = vals[3];
```

The `IlvValue` class converts values automatically as required. This means that you do not need to know the exact type of a value that you set or retrieve. For example, you could set the position of an object using a string value as follows:

```
IlvValue xval("x", "100");
instance->changeValue(xval);
```

Conversely, when you retrieve a value, you can convert it to the type you need as follows:

```
IlvValue xval("x");
instance->queryValue(xval);
float x = xval;
```

User-Defined and Predefined Attributes

A prototype and its instances have three kinds or attributes: user-defined attributes, predefined attributes, and sub-attributes.

User-Defined Attributes

The user-defined attributes are the attributes defined by the accessors that you attached to the prototype when you designed it in IBM ILOG Views Studio. They vary from one prototype to another. The effect of setting or retrieving a user-defined attribute is determined by the accessor objects that compose it.

For example, suppose that you have created a prototype representing a thermometer. You defined a `temperature` attribute by adding a reference accessor that maps the temperature to the `value` attribute of a gauge. To change the temperature displayed by an instance of your prototype, use the `changeValue` method as follows:

```
IlvValue tempVal("temperature");
tempVal = 22.5;
instance->changeValue(tempVal);
```

Predefined Attributes

The predefined attributes of a group let you modify or retrieve common properties that all prototypes have, such as the position, the size, the visibility, and so on.

Most predefined attributes take effect only when a group is added to a manager or a container, but they can be set before that. They are stored in the graphic node but only take effect when the group is added.

The predefined attributes are listed in Table 4.2.

Table 4.2 *Predefined Attributes of Prototypes and Prototype Instances*

Attribute	Type	Description
`layer`	`IlInt`	Set this attribute to move all the nodes of the group to a given layer. Retrieving this attribute returns the layer where the nodes of the group are contained. If all nodes are not in the same layer, the result is undefined.
`visible`	`IlBoolean`	Set this attribute to hide or show a group. Retrieving this attribute returns `IlTrue` if all the graphic nodes of the group are visible and `IlFalse` if they are all invisible. The result is undefined if some nodes are visible and other nodes are invisible.
`x`	`IlInt`	This attribute is the horizontal coordinate of the upper-left corner (in manager coordinates) of the group bounding box, without applying any view transformers.
`y`	`IlInt`	This attribute is the vertical coordinate of the upper-left corner (in manager coordinates) of the group bounding box, without applying any view transformers.
`width`	`IlUInt`	This attribute is the width of the group bounding box (in manager coordinates), without applying any view transformers.
`height`	`IlUInt`	This attribute is the height of the group bounding box (in manager coordinates), without applying any view transformers.
`centerX`	`IlInt`	This attribute is the horizontal coordinate of the center of the group bounding box (in manager coordinates), without applying any view transformers.

Table 4.2 Predefined Attributes of Prototypes and Prototype Instances (Continued)

Attribute	Type	Description
`centerY`	`IlInt`	This attribute is the vertical coordinate of the center of the group bounding box (in manager coordinates), without applying any view transformers.
`interactor`	`const char*`	Set this attribute to associate an interactor with all the graphic nodes of the group. The value of the attribute is the interactor name (for example, `"Button"`). Retrieving this attribute returns the name of the interactor associated with the graphic nodes of the group. If all nodes do not have the same interactor, the result is undefined.

Sub-attributes

The sub-attributes of prototypes let you directly access the attributes of the objects contained in your prototypes. The names of sub-attributes are built by concatenating the path of the object and the attribute name. The components of a sub-attribute name are separated by a period. For example, if your prototype contains an `IlvLabel` named `title`, you can set or retrieve its label using the attribute name `title.label`.

All the predefined properties listed in Table 4.2 can also be accessed for a particular graphic node.

Linking Prototypes to Application Objects

This section describes the three methods that can be used to link prototypes to application objects:

- *Setting Values Directly*: This is the easiest way if you simply want to feed values from your application to the views.
- *Using Group Mediators*: This allows the application to both drive the interface and be notified of value changes produced by the user.
- *Using Proto Mediators*: This enables you to build object factories that will link application classes with prototypes, thereby creating the interface of a dynamic application automatically.

Setting Values Directly

The sample `base_feed` (contained in the `<ILVHOME>/samples/protos` directory) shows how to drive your interface from your application. Once you have downloaded a panel

containing instances of prototypes, or created your instances in a manager or container, you retrieve the instances that you want to edit:

```
IlvGroupHolder* groupHolder= IlvGroupHolder::Get(manager);
IlvGroup* myThermometer= groupHolder->getGroup("thermometer");
```

Then, you change its values with the `IlvGroup::changeValue` method:

```
if (myThermometer)
   myThermometer->changeValue(IlvValue("temperature",(IlUInt) 20)));
```

Using Group Mediators

A group mediator (class `IlvGroupMediator`) is used to connect an object of the application to a prototype and serves as an interactive graphic editor for the object (also called an object inspector). The samples `inspector` and `synoptic` (contained in the `<ILVHOME>/samples/protos` directory) implement a group mediator and can be used as a baseline.

The following code sample shows how to develop an application that cleanly separates the user interface from the application code. Assume that you have an application that includes a `Machine` base class and a `Boiler` specialization class:

```
class Machine { // The base class of most application objects.
protected:
     list<MachineObserver* > observers;
};
class MachineObserver { // A notification mechanism serving as a
                        // generic communication means between objects.
public:
     void observe(Machine* m) { m->observers.append(this); }
     virtual void notify (Machine*);
};
class Boiler : public Machine { // The class for which you want
                                // to create an object inspector.
public:
// Temperature is an attribute you want the user to have control of.
     void set_temperature(float) { ...
             for each observer in observers
                      observer->notify(this);
     }
     float get_temperature();
};
```

These classes perform a simulation, a process control, or any computational activity independent of any kind of interactive or graphic behavior. A group mediator allows you to implement a graphical user interface for the `Boiler` without introducing any dependencies in the application classes, which are assumed to be much more complex.

For this, you want to create a subclass of `IlvGroupMediator` that will handle the graphic representation and the user interaction of a machine of class `Boiler`:

```
class BoilerUI : public IlvGroupMediator,  public MachineObserver {
public:
```

```
     BoilerUI(IlvGroup* ui, Boiler* b) : IlvGroupMediator(ui, b) {
          MachineObserver::observe(b);
          if (!temperatureSymbol)
            temperatureSymbol=IlvGetSymbol("temperature");
     }
     Boiler* boiler() { return (Boiler*) getObject(); }
     void queryValues(IlvValue* vals, IlUInt) const {
            if (vals[0].getName() == temperatureSymbol))
                 vals[0] = boiler()->get_temperature();
     }
     void changeValues(const IlvValue* vals, IlUInt) {
            if (vals[0].getName() == temperatureSymbol))
                  boiler->set_temperature(vals[0]);
     }
     void notify(Machine*) { update(); }
static IlvSymbol* temperatureSymbol;
};
```

This class serves as a bridge between a prototype instance and an application object. It defines four methods:

- The constructor establishes a link and the `observe(b)` statement declares to the application that it wants to be notified of internal changes occurring to the boiler.
- The `changeValue()` method, which is called whenever the user changes an attribute of the object. It notifies the object that it should update its temperature value. It can handle other attributes as well.
- The `queryValue()` method, which is called whenever the prototype needs to update its values. It queries the internal values of the object and transfers them to the user interface.
- The `notify()` method, which must be called explicitly from within the application whenever an internal attribute of the object changes in order for these changes to be reflected in the user interface. Any call to `Boiler::set_temperature()` automatically notifies all observers, which means that the `notify()` method does not need to be called explicitly. Other applications that do not implement an observable/observer design pattern such as this may want to call `notify()` from other parts of the internal code.

Once the mediator class has been defined, you can dynamically link an object of the application to a prototype instance that is used as a boiler inspector:

```
IlvGroup* myBoilerInspector = groupHolder->getGroup("BoilerInspector");
BoilerUI* myBoilerUI = new BoilerUI(myBoilerInspector, myBoiler);
```

You can change the application object being inspected by the prototype at any time:

```
myBoilerUI->setObject(myOtherBoiler);
```

Even though this mechanism requires some application-specific coding, it is very generic—any application data structure can be adapted to use it. Once the mediator class has been designed, the user interface and the application become completely independent entities. Each can be developed and maintained separately. The user interface is developed using

IBM ILOG Views Studio and the application using any application development environment.

The group mediator also has a lock mechanism that can be used to prevent unnecessary refreshes of the user interface. In the above example, the boiler `set_temperature` method calls the `notify()` method of the `BoilerUI` to refresh the user interface. Since the change of values comes from the `UI`, it may be unnecessary to perform this last refresh. Testing the locked flag prevents such refreshes:

```
void BoilerUI::changeValues(const IlvValue* vals, IlUInt) {
    if (locked()) return;
    if (vals[0].getName() == temperatureSymbol))
           boiler->set_temperature(vals[0]);
}
```

Using Proto Mediators

A proto mediator (class `IlvProtoMediator`) is a subclass of `IlvGroupMediator` and is used to dynamically create prototype instances of a given class and place them in a manager or container. The idea is to design a specific prototype for each main application class. When an object is created by the application, a corresponding prototype is instantiated and placed in the manager. This allows you to create a graphical user interface for a complete application, separating the user interface design from the functional core of the application. The following samples from the `<ILVHOME>/samples/protos` directory implement this design pattern: `interact_synoptic` to build an air-traffic control simulator, and `synoptic` to build a simulator for a manufacturing plant.

For example, assuming the same base application (`Machines` and `Boilers`), you want each `Boiler` instance to be represented and edited at the same time by the user. Create a subclass of `IlvProtoMediator`:

```
class BoilerUI: public IlvProtoMediator, public MachineObserver {
public:
    BoilerUI(IlvManager*m,Boiler*b)
             :IlvProtoMediator(m,"BoilerPrototype",b)
        {
          observe(b);
          IlvSymbol* vals[2] = {
              IlvGetSymbol( "x"), IlvGetSymbol("y") };
          update(vals); // Sets the position of the current instance.
          // The application must have a way of specifying where to place
          // the object. Alternatively, you can handle the placement by
          // explicitly setting the x and y values of the BGO.
          install(m); // Place the prototype in the manager
        }
// Other methods are the same as the BoilerUI using the GroupMediator.
};
```

Now, the application can have a global “user interface factory” responsible for generating prototype instances as soon as it creates its internal objects. The code of this factory may look like the following pseudo-code:

```
class myApplication {
      list<Boiler*> boilers;
      void initUI (IlvManager* m) {
           for each machine in boilers
               new BoilerUI(m, machine);
      }
      void add_boiler(Boiler* b) {
          boilers.append(b);
          new BoilerUI(getManager(), b);
      }
};
```

Advanced Uses of Prototypes

This section describes the following advanced topics on using prototypes:

- *Writing New Accessor Classes*
- *Creating Prototypes by Coding*
- *Customizing IBM ILOG Views Studio With the Prototypes Extension*

Writing New Accessor Classes

The Prototypes package contains many predefined accessor classes that allow you to define complex behaviors in your prototypes. You may, however, wish to implement specific behaviors for your particular needs. This section explains how you can extend the set of accessor classes you use to build your prototypes. It also explains how your new accessor classes are integrated into IBM® ILOG® Views Studio.

To add a class of accessors, you simply have to write two classes:

- A subclass of `IlvUserAccessor` that defines the effect of your new accessor.
- A subclass of `IlvAccessorDescriptor` that defines the way your accessor will be edited in IBM ILOG Views Studio.

The `<ILVHOME>/samples` directory of the IBM ILOG Views distribution contains an example of a new accessor class (the `gpacc.h` and `gpacc.cpp` files). See the `README` file in that directory for more information.

Subclassing IlvUserAccessor

To define a new accessor class, you can either write a direct subclass of `IlvUserAccessor` or derive from an existing subclass that implements the features you want to extend. You may also want to make this class persistent.

Defining the Subclass

The declaration of a subclass of `IlvUserAccessor` will typically appear as follows:

```
class MyAccessor: public IlvUserAccessor {
```

```
public:
    MyAccessor(const char* name,
               const IlvValueTypeClass* type,
               const char* param1,
               const char* param2);
    DeclareUserAccessorInfo();
    DeclareUserAccessorIOConstructors(MyAccessor);
protected:
    IlvSymbol* _param1;
    IlvSymbol* _param1;
    virtual IlBoolean changeValue(IlvAccessorHolder* object,
                                  const IlvValue& val);
    virtual IlvValue& queryValue(const IlvAccessorHolder* object,
                                 IlvValue& val) const;
}
```

The following methods must be redefined to create a new accessor class:

- `MyAccessor`

```
MyAccessor(const char* name,
           const IlvValueTypeClass* type,
           const char* param1,
           const char* param2);
```

This constructor is used to create an instance of your accessor by code. In IBM ILOG Views Studio, only the input constructor will be used. The `name` parameter defines the name of the attribute that will be handled by the accessor and the `type` parameter defines the type of the attribute. Your constructor will probably have additional parameters, such as `param1`. These parameters are often character strings that correspond to the parameters that the user can input in IBM ILOG Views Studio and that are evaluated at runtime.

- `changeValue`

```
virtual IlBoolean changeValue(IlvAccessorHolder* object,
                              const IlvValue& val);
```

The `changeValue` method is called when the attribute handled by the accessor is changed using a call to `changeValue` on the prototype or one of its instances. You use this method to define the effect of changing the value of your accessor. If your accessor uses parameters, you must evaluate these parameters. This can be done using the `getValue` method that evaluates a string containing either an immediate value or the name of another accessor.

The `object` parameter is the prototype or the prototype instance to which the accessor is attached. The `val` parameter contains the new value. The `changeValue` method must return `IlTrue` if the value was successfully changed, or `IlFalse` if an error occurred (for example, if one of the parameters could not be evaluated).

- `queryValue`

```
virtual IlvValue& queryValue(const IlvAccessorHolder* object,
                             IlvValue& val) const;
```

The `queryValue` method is called when the attribute handled by the accessor is retrieved using a call to `queryValue` on the prototype or one of its instances. This method must store the "current" value of the accessor in its `val` parameter (if doing so is appropriate). Some accessors store their current value, while others do not (for example, Condition accessors do not store their value). The current value is stored in the `val` parameter using the assignment operator of `IlvValue`. The method must return its `val` parameter.

- `initialize`

  ```
  virtual void initialize(const IlvAccessorHolder* object);
  ```

  The `initialize` method is called when the accessor object is associated with its prototype or prototype instance. You can redefine this method to perform any kind of initialization.

Making the IlvUserAccessor Subclass Persistent

Like graphic objects, accessor objects need to be persistent, which means they are saved to the prototype definition file and are read when the prototype is loaded. The persistence mechanism for accessor objects is very similar to the mechanism used for graphic objects.

First, in the `.h` file of your accessor class, you must call the following macros in the `public` section of the class declaration:

```
DeclareUserAccessorInfo();
DeclareUserAccessorIOConstructors(MyAccessor);
```

This automatically creates the IBM ILOG Views runtime type information for your subclass and declares the persistence and copy methods.

In the `.cpp` file, you then have to write the following methods:

- `MyAccessor(IlvDisplay* display, IlvGroupInputFile& f)`
- `MyAccessor::MyAccessor(const MyAccessor& source)`
- `MyAccessor::write(IlvGroupOututFile& f) const`

This constructor reads the description of your accessor object from an input stream. The `IlvGroupInputFile` class is similar to `IlvInputFile`. Typically, you use only its `getStream` method. This returns a reference to an `istream` object from which you can read the description of your accessor object. However, the convenience method `readValue` can be used. The `writeValue` method puts quotation marks around strings containing spaces, and the `readValue` method checks for these quotation marks and reads the string correctly. Combined use of these methods avoids input/output errors. For example, the implementation of the method could be as follows:

```
MyAccessor::MyAccessor(IlvDisplay* display, IlvGroupInputFile& f)
: IlvUserAccessor(display, f)
{
        _param1 = f.readValue();
        _param2 = f.readValue();
```

```
}
```

Next, you have to write a copy constructor that will be called when the prototype is copied or when an instance of the prototype is created:

```
MyAccessor::MyAccessor(const MyAccessor& source)
: IlvUserAccessor(source)
{
        _param1 = source._param1;
        _param2 = source._param2;
}
```

The `write` method must be redefined to save the description of the accessor. The format used to save the parameters must match the format defined by the input constructor:

```
MyAccessor::write(IlvGroupOututFile& f) const
{
        IlvUserAccessor::write(f);
        f.writeValue(_param1); f << IlvSpc();
        f.writeValue(_param2); f << endl;
}
```

Finally, the following macros must be called in the `.cpp` file:

```
IlvPredefinedUserAccessorIOMembers(MyAccessor)
IlvRegisterUserAccessorClass(MyAccessor,IlvUserAccessor);
```

Subclassing IlvAccessorDescriptor

Once you have written your subclass of `IlvUserAccessor`, you need to write another class, a subclass of `IlvAccessorDescriptor`. This class provides the information needed by the Group Inspector of IBM ILOG Views Studio to edit the parameters of your accessor class.

The name of the `IlvAccessorDescriptor` subclass must match the name of the subclass of `IlvUserAccessor`. For example, if your accessor class is `MyAccessor`, the descriptor class must be called `MyAccessorDescriptorClass`.

You only need to declare the accessor descriptor class. An instance of it will be automatically created and associated with your user accessor subclass by the `IlvRegisterUserAccessorClass` macro.

Here is a typical example of a descriptor class:

```
class MyAccessorDescriptorClass :
public IlvAccessorDescriptor {
public:
    MyAccessorDescriptorClass()
        : IlvAccessorDescriptor("MyAccessor: an example",
                Miscellaneous,
                "example %s %s...",
                IlFalse,
                &IlvValueIntType,
                0,
                2,
```

```
                "Parameter #1", &IlvValueParameterTypeString,
                "Parameter #2", &IlvNodeNameParameterType) {}
};
```

The accessor descriptor class only requires a constructor with no arguments. It must call the `IlvAccessorDescriptor` constructor. For a detailed explanation of the parameters of this constructor, see the description of the `IlvAccessorDescriptor` class.

Creating Prototypes by Coding

Prototypes are meant to be designed graphically using IBM® ILOG® Views Studio. In some cases, however, you may need to create prototypes or to modify existing prototypes from a C++ program. This section explains how you can create prototypes by coding in C++ instead of designing them with IBM ILOG Views Studio.

Creating a New Prototype

A prototype is represented by an instance of the `IlvPrototype` class. To create a new prototype, use the following constructor:

```
IlvPrototype* proto = new IlvPrototype("myPrototype");
```

Adding Graphic Nodes

The first step is to define the graphic appearance of the prototype. This is done by adding nodes containing graphic objects. For this, you create instances of the `IlvGraphicNode` class and add them to the prototype using the `addNode` method.

```
IlvLabel* label = new IlvLabel(display, 100, 100, "Hello");
IlvGraphicNode* node = new IlvGraphicNode(label, "label", IlTrue);
proto->addNode(node);
```

The `IlvGraphicNode` constructor has three parameters:

- An `IlvGraphic`: the graphic object to include in the prototype.
- A string: the name of the node.
- A Boolean: specifies whether a local transformer should be associated with the graphic node. (See the `IlvGraphicNode` class for details.)

You must give different names to the graphic nodes of your prototype if you need to reference them in accessor parameters.

Adding Subgroups

You can create hierarchical objects by adding a subgroup to your prototype. To do this, you must add a node that is an instance of the `IlvSubGroupNode` class. This subgroup can be an `IlvGroup` that you build yourself by adding graphic nodes to it, or it can be an instance of another prototype:

```
// Add a sub-group:
IlvGroup* subgroup = new IlvGroup("subgroup");
```

```
IlvLine* line1 = new IlvLine(display, IlvPoint(100, 100),
                                      IlvPoint(200, 200));
subgroup->addNode(new IlvGraphicNode(line1, "line1"));
IlvLine* line2 = new IlvLine(display, IlvPoint(100, 200),
                                      IlvPoint(200, 100));
subgroup->addNode(new IlvGraphicNode(line2, "line2"));
proto->addNode(new IlvSubGroupNode(subgroup));
// Add a prototype instance as a sub-group:
IlvPrototype* proto = IlvLoadPrototype("samples.pump", display);
IlvProtoInstance* instance = proto->clone();
proto->addNode(new IlvSubGroupNode(instance));
```

Adding Accessor Objects

Once you have “drawn” your prototype by adding graphic objects to it, you can define its properties and specify the effect of changing these properties. To do this, you add accessor objects to your prototype. Accessor objects are instances of subclasses of `IlvUserAccessor`.

To add an accessor object to your prototype, create an instance of the appropriate subclass of `IlvUserAccessor` and call the `addAccessor` method. For example, the following code adds two accessor objects to a prototype: an `IlvValueAccessor` that stores a value and an `IlvConditionAccessor` that tests a condition and changes a attribute according to the result.

```
proto->addAccessor(new IlvValueAccessor("v", IlvValueFloatType));
proto->addAccessor(new IlvConditionAccessor("v", IlvValueFloatType,
                       display,
                       IlvConditionAccessor::IlvCondGreaterThan,
                       "100",
                       "label.label",
                       "Greater than 100",
                       "Smaller than 100"));
```

See the section *Predefined Accessors* and the *IBM ILOG Views Prototypes Reference Manual* for a complete description of each accessor class.

Adding the Prototype to a Library

Prototypes must be stored in a prototype library so that they can be saved and reloaded later.

To create a new prototype library, use the `IlvProtoLibrary` class:

```
IlvProtoLibrary* protoLib = new IlvProtoLibrary(display,
                                "myLib",
                                "/usr/home/myhome/protos");
```

A prototype library stores its prototypes in a file system directory (`"/usr/home/myhome/protos"` in the previous example). You can change this directory later using the `setPath` method.

To add your prototype to the new library, call the `addPrototype` method:

```
protoLib->addPrototype(proto);
```

Saving the Prototype

To save your prototype, call the `IlvAbstractProtoLibrary::save` method:

```
myLib->save(0, IlTrue);
```

The first parameter is an optional output stream where the library description file is saved. Set it to `0` so that the description file is saved to its default location (`"/usr/home/myhome/protos/myLib.ipl"` in the previous example). The second parameter is set to `IlTrue` to specify that all the prototypes must be saved.

Customizing IBM ILOG Views Studio With the Prototypes Extension

This section describes the most important classes that you can derive to extend IBM® ILOG® Views Studio with the Prototypes extension.

Extension Class

The IBM ILOG Views Studio extension is represented by the `IlvStPrototypeExtension` class, which is declared in `<ILVHOME>/studio/ivstudio/protos/stproto.h`:

```
class ILVSTPRCLASS IlvStPrototypeExtension
  : public IlvStExtension {
public:
    IlvStPrototypeExtension(IlvStudio* editor);
    static IlvStPrototypeExtension* Get(IlvStudio* editor);
};
```

An instance of this class (or a subclass) must be created after the `IlvStudio` object is created and before the `initialize` method is called. The static `Get` method returns the (unique) instance of `IlvStPrototypeExtension`.

Buffer Classes

IBM ILOG Views Studio defines four subclasses of `IlvStBuffer`. These classes are also declared in `<ILVHOME>/studio/ivstudio/protos/stproto.h`.

IlvStPrototypeManagerBuffer

The `IlvStPrototypeManagerBuffer` class represents a buffer of the “Prototype Instances (2D)” type. The `NewPrototypeManagerBuffer` command creates an instance of this class. The manager controlled by an `IlvStPrototypeManagerBuffer` is an instance of `IlvManager`:

```
class ILVSTPRCLASS IlvStPrototypeManagerBuffer
  : public IlvStBuffer
{
public:
    IlvStPrototypeManagerBuffer(IlvStudio*,
                                const char* name,
                                IlvManager* = 0);
};
```

IlvStPrototypeEditionBuffer

The `IlvStPrototypeEditionBuffer` class represents a buffer of the “Prototype” type, that is, a buffer used to edit a prototype. The `NewPrototypeEditionBuffer` command creates an instance of this class. The manager controlled by an `IlvStPrototypeEditionBuffer` is an instance of `IlvGadgetManager`:

```
class ILVSTPRCLASS IlvStPrototypeEditionBuffer
  : public IlvStPrototypeManagerBuffer
{
public:
    IlvStPrototypeEditionBuffer(IlvStudio*,
                                const char* name,
                                IlvManager* = 0);
    void editPrototype(IlvPrototype* prototype,
                       IlBoolean fromLib = IlTrue,
    const char* filename = 0);
    IlvPrototype* getPrototype();
    IlvPrototype* getEditedPrototype();
   };
```

The `editPrototype` method initializes the buffer so that it can edit the prototype specified by `prototype`. A copy of the prototype is made and is stored in the associated manager. The `fromLib` argument specifies whether the edited prototype is stored in a prototype library contained in the Prototypes palette or if the prototype is a “standalone” prototype loaded from a `.ivp` file. In the second case, the optional `filename` argument can contain the full path name of the `.ivp` file.

The `getPrototype()` method returns the prototype contained in the buffer. The `getEditedPrototype()` method returns the “original” prototype if the buffer is currently editing a prototype from a library. Otherwise, it returns 0.

C H A P T E R

5

Predefined Accessors

Accessors are basic building blocks that define the value and behaviors of a BGO (`IlvGroup` or `IlvPrototype`). An attribute usually consists of a Data accessor and one or more Control accessors that define its side effects when the attribute is set. This section lists the accessor classes that are predefined in the Prototypes library, and is divided as follows:

- *Overview*
- *Data Accessors*
- *Control Accessors*
- *Display Accessors*
- *Animation Accessors*
- *Trigger Accessors*
- *Miscellaneous Accessors*

Overview

Each accessor class is illustrated by one or more sample prototypes. Most of these samples are contained in one of the prototype libraries included in the IBM® ILOG® Views distribution:

- `<ILVHOME>/data/ivprotos/libs`
- `<ILVHOME>/samples/protos/*/data/*.ipl` subdirectories

To look at a sample prototype:

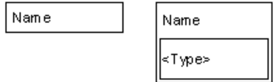
1. Launch IBM ILOG Views Studio with the Prototypes extension.
2. Open the `.ipl` file containing the corresponding prototype library.
3. Double-click on the prototype in the palette.

Graphic Representation of the Behavior of a Prototype

In the examples that illustrate each behavior class, the data flow defined by the accessors of a prototype is represented using the following graphic vocabulary:

- A rectangle represents an accessor (elementary piece of behavior).
- An attribute is represented by a stack of accessors with a given name. In such a stack, the accessors are evaluated from top to bottom when the value of the attribute is changed or queried.
- The order of evaluation is represented by the relative position of an accessor in its stack.
- An inset rectangle is used to represent the type of the given attribute.

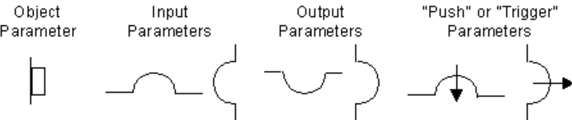
A graphic representing these items is shown here:



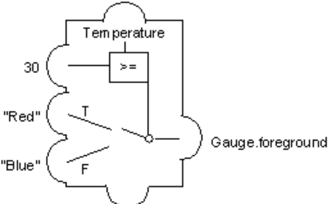
Also:

- Slots on the sides of accessors represent the parameters of the accessor.
- A round slot represents a value parameter.
- A square slot represents an object parameter.
- Slots at the top represent the input access to a value.
- Slots at the bottom represent its output.
- Slots on the left side represent input parameters of the accessors (the accessors will query their value when they are evaluated).
- Slots on the right side represent output parameters (the accessors will change the values).
- Finally, slots with an arrow indicate that the value will be pushed instead of simply set. The arrow is used to indicate Trigger accessors.

A graphic representing these items is shown here:



To complete the model, links or direct values are used to connect the accessor output to other input attributes. The following diagram shows a Condition accessor with these conditions. If Temperature is set to above 30, the foreground of the Gauge object will be set to Red. Otherwise, it will be set to Blue.



Data Accessors

Data accessors hold a value or a pointer to values. They define the type of a given attribute. They are similar to variable declarations in a programming language such as C++. All attributes should contain one of these accessors and no more.

> ***Note:*** *Some accessors also hold a value (Rotate for instance), which means values that hold them do not need an extra Data accessor.*

The different Data accessors are described as follows:

- *Value*
- *Reference*
- *Group*
- *Script*

Value

The Value accessor (class `IlvValueAccessor`) lets you attach an attribute holding a value to a prototype. When the value is modified, it is simply stored. When the value is queried, the last value stored is returned.

Parameters

◆ No parameters, but the type of the value must be specified since it cannot be deduced.

Example:

The `invertedColor` attribute in the `pump` prototype of the `samples` prototype library stores a color name as a temporary variable.

Reference

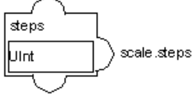
The Reference accessor (class `IlvNodeAccessor`) is used to reference an attribute of one of the prototype nodes (also called sub-attributes) at the prototype level. When the corresponding attribute is changed, the new value is forwarded to the specified sub-accessor. Conversely, when the attribute is queried, it is first queried from the node and forwarded to the prototype. A Reference accessor is similar to a reference (a pointer or an alias) in a programming language.

Parameters

◆ **Accessor**: Node attribute or prototype value that holds the value. The type of the value is determined by what the accessor points to.

Example

The `steps` attribute in the `thermo` prototype of the `samples` library points directly to the `steps` attribute of the `scale` object. When the attribute `steps` is set, it is assigned to the `scale.step` attribute. If the `scale.step` attribute is changed by the program, any query of the attribute returns the new value.



Group

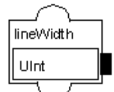
The Group accessor (class `IlvGroupUserAccessor`) defines an attribute that will collectively reference all the sub-attributes of the same name in all group nodes. For example, you can use this accessor with the name `foreground` and the type `Color` to change the foreground color of all the prototype elements in one single assignment.

Parameters

- No parameters. The name of the attribute is used to determine the subattribute that will be referenced by this accessor. The type of the accessor is implicitly determined.

Example

In the `pump` prototype of the `samples` prototype library, a `lineWidth` attribute can be added. The attribute should be of type `UInt`. Changing this attribute from the Group Inspector (using the Attributes notebook page) changes the line width of all the graphic objects that have a `lineWidth` defined.



Script

The Script accessor (`IlvJavaScriptAccessor`) class lets you program the behavior of your prototypes using the scripting language interpreter included in IBM ILOG Views Studio.

A Script accessor has two parameters, which are the names of script functions:

- The `set` function is called when the value of the accessor is changed. It must be of the form:

  ```
  function SetX(obj, newval)
  {
     ...
  }
  ```

  The `obj` argument is the prototype associated with the accessor. The `newval` argument is the new value that has been assigned to the attribute.

- The `get` function is called when the value of the accessor is queried. It must be of the form:

  ```
  function GetX(obj)
  {
     ...
     return(val);
  }
  ```

  The `obj` argument is the prototype associated with the accessor. The function must return a value, which becomes the new value of the attribute.

In the functions associated with a Script accessor, you can access and modify any prototype attribute or a prototype node. Either one of the two function names of the Script accessors can be `none`, in which case no function is called.

The functions associated with a Script accessor can be edited using the IBM ILOG Views Studio Script Editor. They will be saved in a file with a `.ijs` suffix in the same directory and with the same file name as the prototype. Otherwise, they are saved in the prototype file or its library file.

> ***Note:*** *Naming conflicts can occur if you load several prototype instances with the same function names in the same panel. Therefore, it is a good idea to prefix the names of all the prototype script functions with the prototype name they belong to. For instance, in the* `samples.thermo` *prototype, if the* `Temp` *value has a Script accessor, its functions should be called* `SamplesThermoTempGet()` *and* `SamplesThermoTempSet()`.

Parameters

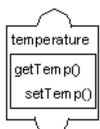
- **Script function (set)**: The name of the script function to execute when the attribute is changed.
- **Script function (get)**: The name of the script function to execute when the attribute is queried.
- The type is determined by the value returned from the `set` function or taken as a parameter by the `get` function. It can, therefore, change dynamically.

Examples

The following function can be used to perform an action similar to a Condition accessor:

```
function SetTemperature(obj, temperature)
{
    if(temperature > obj. threshold) {
         obj.gauge.foreground = "red";
    } else {
         obj.gauge.foreground = "blue";
    }
}

function GetTemperature(obj)
{
    return obj.gauge.foreground;
}
```



Control Accessors

Control accessors perform side effects on other attributes when they are evaluated. They represent the control structures and instructions of a programming language. In the Group Inspector, you can view them from the “Behaviors” and “Interaction” notebook pages, under the “do” clause attached to each attribute.

> ***Note:*** *These accessors are write-only. They do not record the last value tested. If you only define a Control accessor for a value, you will not be able to read this value back. To store the value associated with an accessor, you must define a Value accessor with the same name.*

The different Control accessors are described as follows:

- *Assign*
- *Condition*
- *Format*
- *Increment*
- *Min/Max*
- *Multiple*
- *Notify*
- *Script*
- *Switch*
- *Toggle*

Assign

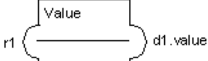
The Assign accessor (class `IlvTriggerAccessor`) is used to assign a value to another attribute or sub-attribute. When the attribute is set, the target attribute specified by the `target` parameter is assigned the specified value.

Parameters

- **Attribute**: Attribute that is modified when this accessor is evaluated.
- **Send**: Attribute or expression that is assigned to Attribute.
- The type of the accessor is undetermined and irrelevant.

Example

The `lcd2` prototype of the `lcd` library uses the Assign accessor.



Condition

The Condition accessor (class `IlvConditionAccessor`) is used to perform a conditional assignment of another attribute when the attribute is changed.

The first parameter defines a condition operator that is applied to the new value of the attribute. For example, if the value of the attribute is changed to `10`, the operator parameter is `>`, and the operand is 5, the condition tested is `10 > 5`. If the operator is `[Operand_value]`, the condition tested is only the value of the operand parameter (that is, the new value passed to `changeValue` is ignored).

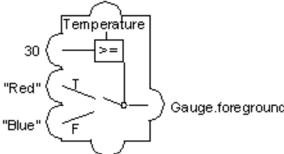
Depending on the test result, the attribute specified by the Attribute parameter is set to one of two values: Value if True or Value if False. The parameters Operand, Value if True, or Value if False can be either immediate values (such as `1` or `"red"`), the names of other attributes that will be queried to get the values used, or an expression containing these immediate values or attribute names.

Parameters

- **Operator**: The operator used to test the conditions. It can be one of the following: `==`, `!=`, `>=`, `<`, `<=`, or `[Operand_value]`.
- **Operand**: The operand value.
- **Attribute**: Prototype value or node attribute that will be set to true or false, depending on the condition.
- **Value if True**: Value to which the output is set if the condition is true (or non `0`).
- **Value if False**: Value to which the output is set if the condition is false (or `0`).
- The type of the accessor is undetermined and irrelevant. However, it needs to be compatible with the operand type.

Example

The following example shows the `thermo` prototype in the `samples` prototype library. If the `temperature` attribute is above 30, the gauge is drawn in red. Otherwise, it is drawn in blue.



Format

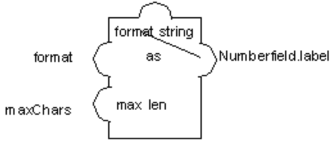
The Format accessor (class `IlvFormatAccessor`) can be used to convert a numeric value of type Double to a character string using a user-specified format. The formatted string is then copied to another accessor. The format of the value is specified by the Format parameter, which is defined in the C library function `printf`. The numeric value is passed to the conversion function as an `IlDouble`, so the format should contain a `%g` specifier.

Parameters

- **Format (printf-style)**: Format string as defined by the `printf` C library function and must be a `String` value. This string must contain at least one `%g`, since this accessor can only convert values of type Double.
- **Max # of chars**: Maximum length of the string after the conversion. If this length is exceeded, the value is replaced by * characters. It must be an Integer value.
- **Attribute**: Attribute to which the formatted value is assigned.

Example

In the `display` prototype in the `samples` prototype library, the Format accessor allows you to change in `NumberField.label` the way the value is displayed.



Increment

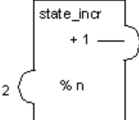
The Increment accessor (class `IlvCounterAccessor)` is used to increment another attribute. Each time the attribute containing this attribute is set, another attribute, called a counter, is increased by one until a specified maximum value is reached. When this value is reached, the counter is reset to zero.

Parameters

- **Maximum**: Maximum value. The value to increase is reset to `0` if it is equal to the maximum value.
- **Attribute**: Attribute to increment.
- The type of the accessor is undetermined and irrelevant.

Example

A three-state button can be implemented by using a Counter accessor linked to a MultiRep accessor. The following accessor has been added to the `symbol` prototype of the `samples` prototype library. Changing the `state_incr` value in the Attributes notebook page of the Group Inspector increments the state and switches its representation.



Min/Max

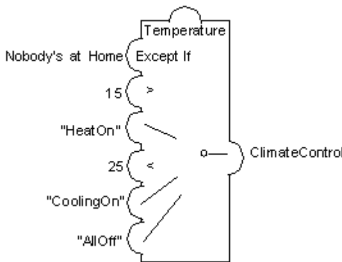
The Min/Max accessor (class `IlvMinMaxAccessor`) is similar to the Condition accessor but handles common cases when an attribute must be tested against a minimum and a maximum threshold. When the attribute is changed, another attribute is set. The assigned value depends on whether the value of the current attribute is less than the minimum, between the minimum and the maximum, or greater than the maximum. In addition, an exception condition can be specified: if the exception condition is true, no value is changed.

Parameters

- **Minimum**: Defines the minimum value.
- **Maximum**: Defines the maximum value.
- **Except if**: If this value is true, the value is ignored and the output value or attribute is not set. The expression must result in a Boolean value.
- **Attribute**: Attribute that is set to one of the following three values.
- **If x < min**: Value to which the attribute is set if the value is less than the minimum.
- **If min < x < max**: Value to which the attribute is set if the value is between the minimum and the maximum.
- **If x > max**: Value to which the attribute is set if the value is greater than the maximum.

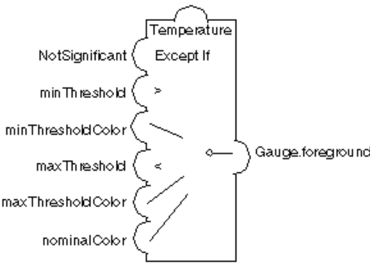
Example 1

This accessor is attached to a `Temperature` attribute. When `Temperature` is set, if `Nobody's at Home` is true, nothing is done. If the `Temperature` is below 15, `HeatOn` is assigned to `ClimateControl`. If `Temperature` is above 25, `CoolingOn` is assigned to `ClimateControl`. If the temperature is between 15 and 25, `AllOff` is assigned to `ClimateControl`.



Example 2

This example shows the `vertGauge` prototype in the `sample` library.



Multiple

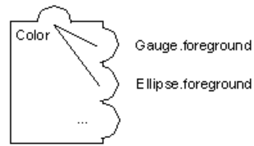
The Multiple accessor (class `IlvCompositeAccessor`) assigns the value of the attribute to multiple other attributes or sub-attributes. It can be used, for example, to change the colors of two graphic nodes using a single public value of the prototype.

Parameters

- ◆ This accessor has a variable number of parameters. Each of these parameters is an attribute or subattribute, to which the value is assigned.
- ◆ All parameters must have compatible types.

Example

This example shows the Color accessor in the `thermo` prototype.



Notify

The Notify accessor (class `IlvOutputAccessor`) turns `changeValue` calls on the attribute to which it is attached into `pushValue` calls. Values that are watching the given attribute will execute all their behaviors.

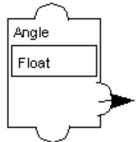
This accessor triggers behaviors of other attributes that depend on the notifying value. For example, you can make a change in the Threshold attribute to also re-evaluate the Temperature attribute. This can be done by attaching a Notify accessor to the Threshold attribute, and a Watch (Threshold) behavior to the Temperature attribute.

Parameters

- No parameters.

Example

The following example shows the `X_Scale` attributes of the `transformer` prototype in the `samples` library.



Script

This accessor is described in Data accessors under *Script*.

Switch

The Switch accessor (class `IlvSwitchAccessor`) implements a switch statement.

Parameters

- **Switch**: An expression that should return an integer. Depending on its result, the attribute `0...N` will be assigned the value of the parameter.
- **case 0**: Must be an attribute of the prototype or the value "". If `Switch` evaluates to `0`, the behaviors of the attribute named in this parameter will be executed.
- **case 1**: If `Switch` evaluates to `1`, the behaviors of the attribute named in this parameter will be executed.
- **...**
- **case N**: If `Switch` evaluates to a value equal to or greater than `N`, the behaviors of the attribute named in this parameter will be executed.

Example

A traffic light with varying settings can be implemented like this:

```
Value    Integer
do
  Switch Value
  case 0 doRed
  case 1 doOrange
  case 2 doGreen
  case 3 Anomaly

doRed
do
  greenEllipse.visible=False
  orangeellipse.visible=False
  redEllipse.visible=True
  doBlink=False

doOrange
do
  greenEllipse.visible=False
  orangeellipse.visible=True
  redEllipse.visible=False
  doBlink=False

doGreen
do
  greenEllipse.visible=True
  orangeellipse.visible=False
  redEllipse.visible=False
  doBlink=False

Anomaly
do
  greenEllipse.visible=False
  orangeellipse.visible=True
  redEllipse.visible=False
  doBlink=true

doBlink Boolean
```

```
do
  blink orangeEllipse.visible 150
```

Toggle

The Toggle accessor (class `IlvToggleAccessor`) switches another attribute between true and false each time the attribute is set. The value assigned to the attribute containing a toggle behavior is ignored.

Parameters

- **Boolean Attribute**: Attribute that is switched when the behavior is evaluated. It must be a Boolean type (for example, the visibility attribute of the object).

Example

The following example shows the `random` prototype in the `sources` prototype library with the value `toggle`.

Display Accessors

Display accessors change the graphic appearance of a node. Ultimately, all accessor networks end up modifying the appearance of the object and, thus, use some kind of Display accessor. General Display accessors such as Rotation, Scale, or Translation change the size and position of a graphic node. One accessor, MultiRep, controls the visibility of nodes, and other accessors, such as Fill, control object-specific properties.

The different Display accessors are described as follows:

- *Fill*
- *MultiRep*
- *Rotation*
- *ScaleX*
- *ScaleY*
- *TranslateX*
- *TranslateY*

Fill

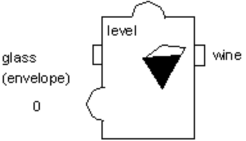
The Fill accessor operates on two polygon objects contained in a prototype: a filled polygon and a filler polygon. The value of the attribute represents a fill level. When the attribute is changed, the points of the filler polygon are modified to fill the polygon to the specified level. An angle can be specified to fill the polygon in any direction.

Parameters

- **Filled Graphic Node**: Must be an `IlvPolygon` graphic node.
- **Filler Graphic Node**: Must be an `IlvPolygon` graphic node.
- **Angle**: A float that indicates the angle at which the fill will be done.

Example

The following is a `bottle` prototype that contains two polygons: the glass and the wine. A Fill accessor is used to define the `level` property. The filled polygon is the glass and the filler polygon is the wine.



MultiRep

The MultiRep accessor (class `IlvMultiRepAccessor`) is used to switch between different representations of a part of your prototype, depending on an integer value. The parameters specify a list of nodes that define the different representations. When the value is changed to `n`, the accessor shows the `n-th` node in the list and hides all the other nodes.

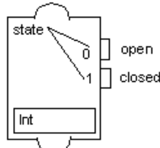
This accessor accepts a variable number of parameters. There are as many representation states as you define rows in the parameter editing matrix. A new row is automatically created in the matrix when you validate the value of the last parameter.

Parameters

- **Graphic Node**: Defines the node that is shown when the value is `0`. Must be a graphic node.
- **Graphic Node**: Defines the node that is shown when the value is `1`. Must be a graphic node.
- The type of this value is `Int` (Integer).

Example

The `symbol` prototype of the `samples` prototype library uses two lines, open and closed, to display a two-state switch.



Rotation

The Rotation accessor (class `IlvRotationAccessor`) lets you set the rotation angle of an object to a given value. The value defined by this accessor is the angle (in degrees) to which the rotation must be set. The angle is stored every time it is set so resetting the value rotates the object by the angle corresponding to the delta between the old and new angles.

The Minimum Angle, Angle Range, Minimum Value, and Value Range parameters are used to compute the new rotation angle given to the input value. The new rotation is computed from the value assigned to the Rotation accessor using the following formula:

```
angle = minAngle + (value - minimum)* Anglerange / range
```

The initial value of the rotation angle is assumed to be the value of the Minimum Angle parameter so the initial position of the rotating object must correspond to this value.

> ***Note:*** *Not all graphic objects are sensitive to rotation. Rectangles, ellipses, and text objects do not rotate. It is recommended to use polygons and splines instead.*

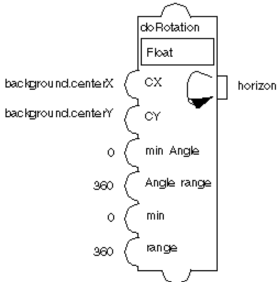
Parameters

- **Graphic Node**: Name of the node to rotate. It must be a graphic node.
- **Center X**: X-coordinate of the rotation center. You can use the centerX accessor for this parameter (Float or Integer).
- **Center Y**: Y-coordinate of the rotation center. You can use the centerY accessor for this parameter (Float or Integer).
- **Minimum Angle**: Minimum angle used to compute the rotation (Float or Integer).
- **Angle Range**: Angle range used to compute the rotation (Float or Integer).
- **Minimum**: Minimum value used to compute the rotation (Float or Integer).
- **Range**: Value range used to compute the rotation (Float or Integer).

- **Handle Interaction**: Boolean specifying whether the accessor should behave like an Event accessor when the user clicks on the node to rotate it. If it is set to true, the user can rotate the node and the accessor value is updated accordingly.
- The type of this value is Float (the angle of rotation).

Example

The following example shows a Rotation accessor attached to the `transformer` prototype in the `samples` library.



ScaleX

The ScaleX accessor (class `IlvZoomXAccessor`) lets you set the horizontal scaling factor of an object. When the value of this accessor is changed, the object is scaled based on the new value. The scaling factor is stored every time it is set so resetting the scale to a different value scales the object by the delta of the old and new scaling factors.

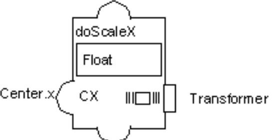
> ***Note:*** *Not all graphic objects are sensitive to the scaling factor. For example, text objects cannot be scaled.*

Parameters

- **Graphic Node**: Name of the graphic node to scale. It must be a graphic node.
- **Center X**: X-coordinate of the center of the scale (Float or Integer).
- The type of this value is Float.

Example

This example shows a Scale accessor attached to a transformer object. The full prototype using this accessor is the `transformer` prototype in the `samples` library.



ScaleY

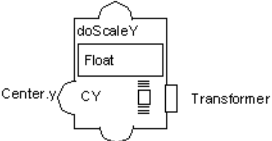
The ScaleY accessor (class `IlvZoomYAccessor`) lets you set the vertical scaling factor of an object. When the value of this is changed, the object is scaled based on the new value. The scale is stored every time it is set, so resetting the scaling factor to a different value changes the size of the object by the delta of the old and new scaling factors.

Parameters

- **Graphic Node**: Name of the graphic node to scale. It must be a graphic node.
- **Center Y**: Y-coordinate of the center of the scale.
- The type of this value is Float.

Example

This example shows a Scale accessor attached to a transformer object. The full prototype using this accessor is the `transformer` prototype in the `samples` library.



TranslateX

The TranslateX accessor (class `IlvSlideXAccessor`) moves a node horizontally to a position determined by a minimum position, a position range, a minimum value, and a value range. The new position is computed from the value assigned to the TranslateX accessor using the following formula:

```
x = xmin + (v - minimum) * xrange / range
```

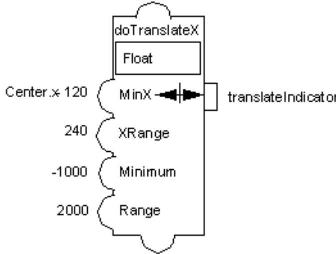
Parameters

- **Graphic Node**: Name of the node to move. It must be a graphic node.
- **Minimum X**: Name of the minimum position (Float or Integer).

- **X range**: Name of the position range (Float or Integer).
- **Minimum**: Name of the minimum value (Float or Integer).
- **Range**: Name of the value range (Float or Integer).
- **Handle Interaction**: Boolean specifying whether the accessor should behave like an Event accessor when the user clicks on the node to rotate it. If it is set to true, the user can rotate the node and the accessor value is updated accordingly.
- The type of the value is Float.

Example

The use of Translate accessors is similar to the use of Scale accessors, except that Translate accessors change the position instead of the size of an object. See the `transformer` prototype in the `samples` library.



TranslateY

The TranslateY accessor (class `IlvSlideYAccessor`) moves a node vertically to a position determined by a minimum position, a position range, a minimum value, and a value range. The new position is computed from the value assigned to the TranslateY accessor using the following formula:

```
y = ymin + (v - minimum) * yrange / range
```

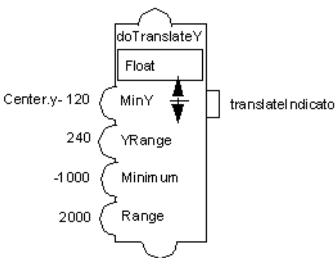
Parameters

- **Graphic Node**: Name of the node to move. It must be a graphic node.
- **MinimumY**: Minimum position (Float or Integer).
- **Y range**: Position range (Float or Integer).
- **Minimum**: Minimum value (Float or Integer).
- **Range**: Value range (Float or Integer).

- **Handle Interaction**: Boolean specifying whether the accessor should behave like an Event accessor when the user clicks on the node to rotate it. If it is set to true, the user can rotate the node and the accessor value will be updated accordingly.
- The type of the value is Float (the distance of translation).

Example

The use of TranslateX and TranslateY is similar to the use of scaleX and scaleY accessors.



Animation Accessors

Animation accessors (class `IlvAnimationAccessor`) are a category of the Display accessors that change the appearance of an object periodically. Animation accessors hold a value of a Boolean type indicating whether the animation is on.

For efficiency reasons, the Animation accessors do not reevaluate their attributes at each count of the timer. Thus, if you change one of the attributes of the accessor, you must reassign the value to itself to force an update of the parameters, using the Assign accessor for instance. See the `pump` prototype in the `samples` library for an example.

The different Animation accessors are described as follows:

- *Blink*
- *Invert*
- *Rotate*

Blink

The Blink accessor (class `IlvBlinkAccessor`) makes an object of your prototype blink, that is, it causes the object to appear and disappear at brief, regular intervals. When the attribute is set to `IlTrue`, the object starts blinking. When the attribute is set to `IlFalse`, the blinking stops.

Parameters

- **Boolean Attribute**: Object attribute that controls the object visibility.
- **Period (ms)**: The interval in milliseconds between two *blinks* (Float or Integer).
- The type of this value is Boolean.

Example

The following example shows the `file` prototype in the `sources` library with a blink value.

Invert

The Invert accessor (class `IlvInvertAccessor`) inverts the color of an element of your prototype periodically. When the property is set to `IlTrue`, the color inversion begins. When the attribute is set to `IlFalse`, the color inversion stops.

While the colors are designated as the foreground and background colors, any colors defined by the prototype or one of its nodes can be used.

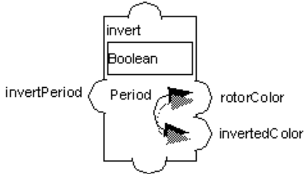
Parameters

- **Fg Col. Attribute**: Node attribute or prototype value that contains the foreground color.
- **Bg Col. Attribute**: Node attribute or prototype value that contains the background color.
- **Period (ms)**: The interval, in milliseconds, between two inversions of the object colors (Float or Integer).
- Type: Boolean (whether the accessors are exchanging their values).

Example

This example is presented in the `pump` prototype of the `samples` prototype library. When `invert` is set to true, the values of `rotorColor` and `invertedColor` are exchanged periodically. The period is defined by the `invert` attribute.

> ***Note:*** *The* `invertPeriod` *value has an Assign behavior:* `invert = invert`. *This forces the accessors to be reevaluated and the internal timer to update its period whenever the period is changed.*



Rotate

The Rotate accessor (class `IlvRotateAccessor`) defines a behavior that, when set to `IlTrue`, makes an object rotate periodically.

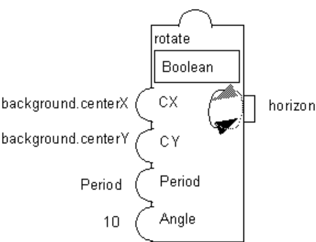
The Angle parameter specifies the number of degrees by which the object rotates at every timer tick. The Center X and Center Y parameters define the rotation center. You should not use the center of the rotating node itself for these parameters because the rounding problems that occur while rotating an object might move it slightly. Instead, you should use the center of another fixed object of the prototype. You can make this reference object invisible if necessary.

Parameters

- **Graphic Node**: Name of the node to rotate. Can be a graphic node or a subgroup node.
- **Angle**: Angle in degrees by which the object is rotated at each step (Float or Integer).
- **Center X**: X-coordinate of the rotation center. You can use the centerX accessor for this parameter (Float or Integer).
- **Center Y**: Y-coordinate of the rotation center. You can use the centerY accessor for this parameter (Float or Integer).
- **Period (ms)**: The interval in milliseconds at which the object rotates. It must be an Integer.

Example

This example is presented in the `pump` prototype of the `samples` prototype library. When the Rotate accessor is set to true, the nodes will turn by 20 degrees every 10 ms.



Trigger Accessors

Trigger accessors define the entry points of evaluation sequences in the graph of accessors. Triggers are accessors that can react to a user event (callback and event), a change in a node by the application (by means of the `pushValue` method), or some other node change (a combination of Trigger and Connect).

The different Trigger accessors are described as follows:

- *Callback*
- *Clock*
- *Watch*
- *Event*

Callback

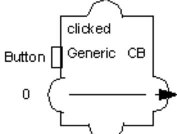
This accessor (class `IlvCallbackAccessor`) attaches a trigger that is set when the given callback is called from a user action on the specified graphic node. For a callback to be called, the node must be either an `IlvGadget` or an `IlvGraphic` to which an interactor has been attached.

Parameters

- **Graphic Node**: The name of the graphic node whose callback is triggered.
- **Callback Name**: The name of the callback.
- **Input**: The value that is sent when the callback is triggered.

Example

The following example shows the `random` prototype in the `sources` library with the `clicked` value. The `clicked` value pushes `0` to its output when the button is pressed by the user. This output is connected to the `toggle` value, which in turn switches the `running` value.



Clock

The Clock accessor (class `IlvAnimationAccessor`) triggers its attribute periodically, executing the attached behaviors. When set to 0, this accessor has no behavior. When set to another value, this value is used as the period of an internal timer that triggers the behavior periodically.

Parameters

- The type of this value is `UInt`. If non-zero, the attribute will not have any effect. Otherwise, its value is interpreted as a timer period.

Watch

The Watch accessor (class `IlvLoopbackAccessor`) makes its attached attribute observe another notifying attribute.

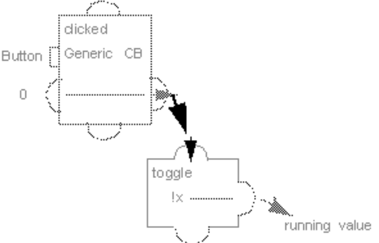
This accessor class is often used with the Callback accessor to change a value of the prototype when a callback is triggered. The Watch accessor connects the triggering attribute containing the callback to the watching attribute that must be changed.

Parameters

- **Notifying Attribute**: Attribute that is observed. This attribute must be one of the attributes that has a Notify or a Callback accessor.

Example

The Watch clicked accessor links the `clicked` value to the `toggle` value, which allows the `running` attribute to be switched whenever the user presses the button attached to the `clicked` value.



Event

The Event accessor (class `IlvEventAccessor`) is used to trigger a behavior in response to user or other application events. When an event of a given type occurs while the mouse pointer is over a node of the prototype, the attribute to which the accessor is attached is evaluated, that is, all its behaviors are set.

While the callback pushes its value to another attribute of the prototype, the event notifies its own attached attribute. The event is thus similar to attaching a Watch accessor to itself after attaching a callback.

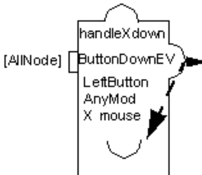
Parameters

- **Graphic Node**: The name of a graphic node. Events received from the input devices are sent to the accessor over this graphic node to trigger a behavior. The special value `[All Nodes]` indicates that this value is triggered when any event of the given type reaches any of the graphic nodes of the prototype.
- **Event Type**: The event type that triggers the accessor. The type can be any of the standard IBM ILOG Views event types: `AnyEvent`, `KeyUp`, `KeyDown`, `ButtonDown`, `ButtonUp`, `EnterWindow`, `LeaveWindow`, `PointerMoved`, `ButtonDragged`, `Repaint`, `ModifyWindow`, `Visibility`, `MapWindow`, `UnMapWindow`, `Reparent`, `KeyboardFocusIn`, `KeyboardFocusOut`, `DestroyWindow`, `ClientMessage`, and `DoubleClick`.
- **Detail**: The detail of the event. This parameter indicates additional filtering of the events and depends on the event type. For example, for a `ButtonDown` event, the detail can be: `AnyButton`, `LeftButton`, `RightButton`, `MiddleButton`, `Button4`, or `Button5`. For a `KeyDown` event, the detail parameter indicates the key, or `AnyKey` that triggers the accessor. See the `IlvEvent` class for a list of the valid keys.
- **Modifiers**: Indicates which modifiers should be pressed. Possible values are: `AnyModifier`, `NoModifier`, `Shift`, `Ctrl`, `Meta`, `Alt`, `Num`, `Lock`, `Alt+G`, or any combination of the previous modifiers such as `Shift+Crtl`, `Ctrl+Shift+Alt`, and so on.

◆ **Event data to send**: The Event attribute that is pushed to the current value. It can be the Type, Detail, X (the horizontal position of the mouse relative to the window), Y (the vertical position of the mouse relative to the window), GlobalX, or GlobalY (the position of the mouse relative to the screen).

Example

The following example shows the `transformer` prototype with an `EventScaleY` value in the `samples` library:



Miscellaneous Accessors

These accessors do not fit current existing categories.

The different Miscellaneous accessors are described as follows:

◆ *Debug*

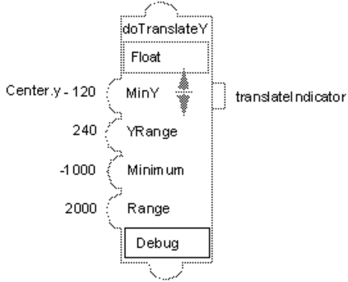
◆ *Prototype*

Debug

The Debug accessor (class `IlvDebugAccessor`) is used to debug prototypes. It prints a message to the console or the output window when the corresponding value is modified or queried.

Example

When the following doTranslateY accessor is queried or changed, a message is printed to the output console, displaying the current value:



Prototype

The Prototype accessor (class `IlvPrototypeAccessor`) allows a new prototype to inherit from all the accessors of an existing prototype. The new prototype behaves as if all the accessors of an existing prototype were added to it. This is useful when building libraries of complex behaviors and in reusing them in other prototypes. The prototype library containing the prototype must be open in order for any instance using this accessor to work properly. From the Group Inspector in IBM ILOG Views Studio, you can add a Prototype accessor to a prototype by selecting the Attributes tab and choosing the Edit>Delegate to Prototype item.

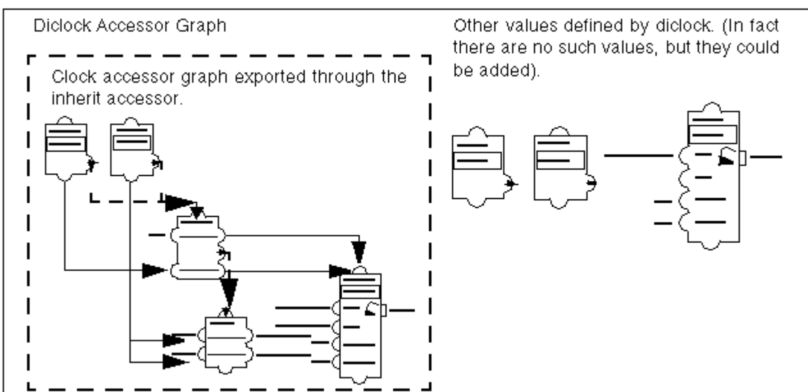
Parameters

- **Prototype name**: Name of the prototype that you want to inherit accessors from.

Example

This accessor is represented as a subgraph showing all the values exported in the context of the current accessor graph.

The `diclock` prototype of the `sources` library encapsulates and exports all the accessors of the `clock` prototype. It behaves exactly like the `clock` prototype but has a different graphic representation.


Diclock Accessor Graph
Clock accessor graph exported through the inherit accessor.
Other values defined by diclock. (In fact there are no such values, but they could be added).

Index

Numerics

2D Graphics buffer window
  description of **74**, **137**

A

abortReDraws member function
  IlvManager class **33**
accelerators
  and managers **53**, **57**
  example in managers **53**
  predefined in managers **53**
accept member function
  IlvMakePolyLinkInteractor class **103**
acceptFrom member function
  IlvMakeLinkInteractor class **103**
acceptTo member function
  IlvMakeLinkInteractor class **103**
accessor objects **152**
accessors **112**
  types **153**
  writing new classes **169**
addAccelerator member function
  IlvManager class **53**
addCommand member function
  IlvManager class **66**
addGhostNode member function
  IlvGrapher class **87**
addLink member function
  IlvGrapher class **87**
addNode member function
  IlvGrapher class **87**
addPoints member function
  IlvPolylineLinkImage class **96**
addTransformer member function
  IlvManager class **22**
addView member function
  IlvManager class **21**
addVisibilityFilter member function
  IlvManagerLayer class **26**
afterDraw member function
  IlvManagerViewHook class **61**
afterExpose member function
  IlvManagerViewHook class **61**
align member function
  IlvManager class **31**
animation accessors
  description **197**
applyInside member function
  IlvManager class **28**
applyIntersects member function
  IlvManager class **27**, **28**
applyToInside member function
  IlvManager class **27**
applyToObject member function
  IlvManager class **27**, **28**, **34**
applyToObjects member function
  IlvManager class **27**
applyToSelections member function

IlvManager class **27**, **50**
applyToTaggedObjects member function
IlvManager class **27**
arc offset
description **95**
fixed value **95**
proportional value **95**
ArcLinkImage mode **77**
arcs **95**
attributes
connecting **156**
predefined **163**
sub-attribues **165**
user-defined **163**

B

beforeDraw member function
IlvManagerViewHook class **61**
behaviors
attributes **152**
graphic representation **179**
input parameters **126**
prototype graphic behaviors **122**
prototype interactive behaviors **127**
binding views **20**
bufferedDraw member function
IlvManager class **32**
business graphic objects
description of **109**

C

C++
prerequisites **12**
changeLink member function
IlvGrapher class **87**
CloseProtoLibrary command **143**
commands
and managers **20**
computePoints member function
IlvLinkImage class **90**, **97**
connecting attributes **156**
connection pins **75**
coordinates **98**
description **98**
editing **104**
managing **98**
providing a faster implementation **99**
recovering the index **98**
returning the unused pin **99**
containers
displaying groups and instances **155**
contentsChanged member function
IlvManager class **68**
IlvManagerViewHook class **61**
control accessors
description **184**
ConvertProtoManager command **143**
copy member function
IlvTranslateObjectCommand class **68**
createLink member function
IlvMakeLinkInteractor class **103**
IlvMakeLinkInteractorFactory class **103**
createNode member function
IlvMakeNodeInteractor class **102**
IlvMakeNodeInteractorFactory class **102**
creating
prototype instances **131**
prototype library **115**

D

data accessors
description **180**
data flow programming **112**
DeletePrototype command **144**
deleteSelections member function
IlvManager class **29**
deSelectAll member function
IlvManager class **29**
display accessors
description **191**
displaying groups and instances **155**
displaying objects
and managers **30**
drawing **32**
doIt member function
IlvDragRectangleInteractor class **43**
IlvMakeRectangleInteractor class **44**

IlvMoveInteractor class **49**
IlvTranslateObjectCommand class **67**
double-buffering
and managers **22**
description **20**
DoubleLinkImage mode **77**
DoubleSplineLinkImage mode **77**
draw member function
IlvManager class **32**
drawGhost member function
IlvDragRectangleInteractor class **43**
IlvMoveInteractor class **48**
drawSpline member function
IlvPolylineLinkImage class **96**
duplicate member function
IlvManager class **32**

E

editing
prototype instances **132**
editing modes
ArcLinkImage **77**
DoubleLinkImage **77**
DoubleSplineLinkImage **77**
group connection **140**
LinkImage **77**
OneLinkImage **77**
OneSplineLinkImage **77**
OrientedArcLinkImage **77**
OrientedDoubleLinkImage **78**
OrientedDoubleSplineLinkImage **78**
OrientedLinkImage **78**
OrientedOneLinkImage **78**
OrientedOneSplineLinkImage **78**
OrientedPolylineLinkImage **78**
PolylineLinkImage **78**
editing modes toolbar **140**
EditPrototype command **144**
end node **89**
end points
position **93**
ensureVisible member function
IlvManager class **22**
events
and accelerators **19**
and interactors **18**
and managers **18**

F

File Menu Commands **139**
fitToContents member function
IlvManager class **22**
fitTransformerToContents member function
IlvManager class **22**, **53**
forgetUndo member function
IlvManager class **66**

G

geometric transformations
and managers **18**
and views **18**
getAccelerator member function
IlvManager class **53**
getCardinal member function
IlvGrapherPin class **98**
getClosest member function
IlvGrapherPin class **99**
getInteractor member function
IlvManager class **37**
getLinkLocation member function
IlvGrapherPin class **99**
getLinkPoints member function
IlvLinkImage class **90**, **97**
IlvPolylineLinkImage class **96**
getLinks member function
IlvGrapher class **88**
getPinIndex member function
IlvGrapherPin class **98**
getSelections member function
IlvManager class **29**
getTo member function
IlvLinkImage class **89**
getViews member function
IlvManager class **21**
ghost images
drawing **101**
global functions

IlvGetContentsChangedUpdate **68**
IlvSetContentsChangedUpdate **68**
grapher
overview **87**
Grapher buffer window
description of **74**, **137**
graphic objects
and managers **28**
selecting in manager **28**
transforming **87**
graphs
loading **88**
managing **86**
querying the topology **88**
saving **88**
grids
and managers **64**
example **65**
snapping **64**
group connection mode **140**
group inspector
description of **142**
group member function
IlvManager class **31**
grouping
and managers **31**
GroupIntoGroup command **144**

H

handleEvent member function
IlvDragRectangleInteractor class **40**
IlvMoveInteractor class **46**
handles
description **92**
hooks **20**, **60**

I

icons
group connection **140**
IlvAbstractProtoLIbrary class **175**
IlvAccessor class **152**
IlvAccessorDescriptor class **172**
IlvAnimationAccessor class **197**, **201**
IlvArcLinkImage class
setFixedOffset member function **95**
setOffsetRatio member function **95**
IlvBlinkAccessor class **197**
IlvCallbackAccessor class **200**
IlvCompositeAccessor class **188**
IlvConditionAccessor class **185**
IlvContainer class **19**, **74**, **111**, **137**
read method **158**
readFile method **158**
IlvCounterAccessor class **186**
IlvDebugAccessor class **203**
IlvDoubleLinkImage
description **94**
IlvDoubleLinkImage class
setFixedOrientation member function **94**
IlvDoubleSplineLinkImage class **94**, **97**
IlvDragRectangleInteractor class **38**
doIt member function **43**
drawGhost member function **43**
handleEvent member function **40**
IlvEllipse class **38**
IlvEvent class **202**
IlvEventAccessor class **202**
IlvFilledRectangle class **38**
IlvFilledRoundRectangle class **38**
IlvFormatAccessor class **186**
IlvGadgetManagerInputFile class **35**
IlvGadgetManagerOutputFile class **35**
IlvGenericPin class
adding connection pins **99**
description **99**
IlvGetContentsChangedUpdate global function **68**
IlvGrapher API **88**
IlvGrapher class
addGhostNode member function **87**
addLink member function **87**
addNode member function **87**
changeLink member function **87**
constructor **87**
description **101**
getLinks member function **88**
isLinkBetween member function **88**
isNode member function **87**
makeLink member function **87**

makeNode member function **87**
mapLinks member function **88**
nodeXPretty member function **88**
nodeYPretty member function **88**
IlvGrapherPin class
description **98**
getCardinal member function **98**
getClosest member function **99**
getLinkLocation member function **99**
getPinIndex member function **98**
setPinIndex member function **98**
IlvGraphic class **87**
description **17**
scale member function **28**
translate member function **28**
IlvGraphicNode class **151**, **173**
IlvGraphicSet class **31**
IlvGraphInputFile class
description **88**
readObject member function **88**
IlvGraphOutputFile class **88**
saving files **88**
writeObject member function **88**
IlvGraphOutputfile class
writeObject member function **88**
IlvGraphSelectInteractor class
constructor **101**
description **101**
IlvGroup class **151**
changeValue method **152**
description **152**
queryValue method **152**
IlvGroupHolder class **111**, **155**
IlvGroupMediator class **166**
IlvGroupNode class **151**
IlvGroupUserAccessor class **181**
IlvInputFile class **34**
IlvInteractor class **18**
IlvInvertAccessor class **198**
IlvJavaScriptAccessor class **182**
IlvLabel class **165**
IlvLayerVisibilityFilter class
isVisible member function **26**
IlvLine class **29**
IlvLineHandle class **29**
IlvLinkHandle class
constructor **92**
description **91**
reference to **87**
IlvLinkImage class
accessing values **89**
computePoints member function **90**, **97**
computing endpoints **90**
constructor **89**
creating custom **96**
description **87**, **89**
getLinkPoints member function **90**, **97**
getTo member function **89**
purpose **89**
setOriented member function **89**
setTo member function **89**
subclassing **90**
IlvLinkLabel class
description **92**
setLabel member function **92**
IlvLoadPrototype class **159**
IlvLoopbackAccessor class **201**
IlvMakeArrowInteractor class **39**
IlvMakeBitmapInteractor class **39**
IlvMakeDoubleLinkImageInteractor class **103**
IlvMakeDoubleSplineLinkImageInteractor class **103**
IlvMakeEllipseInteractor class **38**
IlvMakeFilledEllipse class **39**
IlvMakeFilledEllipseInteractor class **38**
IlvMakeFilledRectangleInteractor class **38**
IlvMakeFilledRoundRectangleInteractor class **38**
IlvMakeLabelLinkImageInteractor class **103**
IlvMakeLineInteractor class **39**
IlvMakeLinkImageInteractor class **103**
IlvMakeLinkInteractor class
acceptFrom member function **103**
acceptTo member function **103**
createLink member function **103**
description **102**
predefined subclasses **103**
setFactory member function **103**
IlvMakeLinkInteractorFactory class
createLink member function **103**

subtyping **103**
IlvMakeNodeInteractor class
createNode member function **102**
description **102**
setFactory member function **102**
IlvMakeNodeInteractorFactory class
createNode member function **102**
subtyping **102**
IlvMakeOneLinkImageInteractor class **103**
IlvMakeOneSplineLinkImageInteractor class **103**
IlvMakePolylineLinkInteractor class **104**
IlvMakePolyLinkInteractor class
accept member function **103**
description **103**
makeLink member function **103**
IlvMakeRectangleInteractor class **38**
description **38**
doIt member function **44**
IlvMakeReliefDiamondInteractor class **38**
IlvMakeReliefLineInteractor class **39**
IlvMakeReliefNodeInteractor class **102**
IlvMakeReliefRectangleInteractor class **38**
IlvMakeRoundRectangleInteractor class **38**
IlvMakeShadowNodeInteractor class **102**
IlvMakeUnZoomInteractor class **39**
IlvMakeZoomInteractor class **39**
IlvManager class **74**, **111**, **137**
abortReDraws member function **33**
addAccelerator member function **53**
addCommand member function **66**
addTransformer member function **22**
addView member function **21**
align member function **31**
applyInside member function **27**, **28**
applyIntersects member function **27**, **28**
applyToObject member function **27**, **28**, **34**
applyToObjects member function **27**
applyToSelections member function **27**, **50**
applyToTaggedObjects member function **27**
bufferedDraw member function **32**
contentsChanged member function **68**
deleteSelections member function **29**
description **87**
deSelectAll member function **29**
draw member function **32**
duplicate member function **32**
ensureVisible member function **22**
fitToContents member function **22**
fitTransformerToContents member function **22**, **53**
forgetUndo member function **66**
getAccelerator member function **53**
getInteractor member function **37**
getSelection member function **29**
getSelections member function **29**
getViews member function **21**
group member function **31**
initReDraws member function **33**
installEventHook member function **37**
installViewHook member function **61**
interactors **101**
invalidateRegion member function **33**
isDoubleBuffering member function **22**
isInvalidating member function **33**
isModified member function **68**
isSelected member function **28**
isUndoEnabled member function **66**
makeColumn member function **31**
makeRow member function **31**
moveObject member function **28**
numberOfSelections member function **29**
read method **158**
readFile method **158**
reDo member function **66**
reDraw member function **32**
reDrawViews member function **33**
removeAccelerator member function **53**
removeEventHook member function **37**
removeInteractor member function **37**
removeView member function **21**
removeViewHook member function **61**
reshapeObject member function **28**
rotateView member function **22**
sameHeight member function **31**
sameWidth member function **31**
setBackground member function **23**
setDoubleBuffering member function **22**
setInteractor member function **37**
setMakeSelection member function **29**

setModified member function **68**
setNumLayer member function **24**
setSelected member function **28**
setTransformer member function **22**
setUndoEnabled member function **66**
shortCut member function **41**, **53**
snapToGrid member function **65**
translateObject member function **28**
translateView member function **22**
unDo member function **66**
unGroup member function **31**
view transformation member functions **22**
zoomView member function **22**
IlvManagerclass
aligning objects member functions **31**
main features **19**
IlvManagerCommand class
advanced features **66**
example **67**
instances **19**
IlvManagerEventHook class **37**
IlvManagerGrid class **64**
IlvManagerInputFile class
description **34**
readObject member function **35**
IlvManagerLayer class
addVisibilityFilter member function **26**
setAlpha member function **26**
IlvManagerObjectInteractor class **52**
IlvManagerOutputFile class
description **34**
example **35**
writeObject member function **35**
IlvManagerViewHook class
afterDraw member function **61**
afterExpose member function **61**
beforeDraw member function **61**
contentsChanged member function **61**
description **61**
interactorChanged member function **61**
transformerChanged member function **61**
viewRemoved member function **61**
viewResized member function **61**
IlvManagerViewInteractor class **18**, **37**, **45**
IlvMinMaxAccessor class **187**
IlvMoveInteractor class
doIt member function **49**
drawGhost member function **48**
example **45**
handleEvent member function **46**
IlvMultiRepAccessor class **192**
IlvNodeAccessor class **181**
IlvOneLinkImage
description **92**, **95**
IlvOneLinkImage class
reference to **93**
setOrientation member function **93**
IlvOneSplineLinkImage class
description **93**
setControlPoint member function **93**
IlvOutputAccessor class **189**
IlvOutputFile class
description **34**
IlvPinEditorInteractor class **104**
IlvPointPool class **90**
IlvPolylineLinkImage class
addPoints member function **96**
description **96**
drawSpline member function **96**
getLinkPoints member function **96**
movePoints member function **96**
reference to **104**
removePoints member function **96**
setPoints member function **96**
IlvProtoGraphic class **111**, **155**
IlvProtoInstance class **155**
IlvProtoLibrary class **155**, **159**, **174**
IlvProtoMediator class **168**
IlvPrototype class **155**, **173**
IlvPrototypeAccesssor class **204**
IlvPrototypeInstance class **160**
IlvRectangle class **44**
IlvReliefDiamond class **38**
IlvReliefLabel class **102**
IlvRotateAccessor class **199**
IlvRotationAccessor class **193**
IlvRoundRectangle class **38**
IlvSCGrapherRectangle **78**
IlvSelectInteractor class **39**, **101**
IlvSetContentsChangedUpdate global function **68**

IlvShadowLabel class **102**
IlvSlideXAccessor class **195**
IlvSlideYAccessor class **196**
IlvStPrototypeEditionBuffer class **176**
IlvStPrototypeExtension class **175**
IlvStPrototypeManagerBuffer class **175**
IlvSubGroupNode class **152**, **173**
IlvSwitchAccessor class **189**
IlvTextField class **62**
IlvToggleAccessor class **191**
IlvTranslateObjectCommand class
  copy member function **68**
  description **67**
  doIt member function **67**
  unDo member function **67**
IlvTriggerAccessor class **184**
IlvUserAccessor class **158**, **169**, **174**
IlvValue class **161**
IlvValueAccessor class **180**
IlvZoomXAccessor class **194**
IlvZoomYAccessor class **195**
initReDraws member function
  IlvManager class **33**
inspectors
  prototype **142**
installEventHook member function
  IlvManager class **37**
installViewHook member function
  IlvManager class **61**
interactorChanged member function
  IlvManagerViewHook class **61**
interactors
  description **101**
  drawing ghost images **101**
  view **37**
invalidateRegion member function
  IlvManager class **33**
isDoubleBuffering member function
  IlvManager class **22**
isInvalidating member function
  IlvManager class **33**
isLinkBetween member function
  IlvGrapher class **88**
isModified member function
  IlvManager class **68**
isNode member function
  IlvGrapher class **87**
isSelected member function
  IlvManager class **28**
isUndoEnabled member function
  IlvManager class **66**
isVisible member function
  IlvLayerVisibility class **26**

L

layers
  and managers **18**, **23**
  default number **24**
  description **17**
  object selectability **25**
  object visibility **25**
  setting up **24**
LinkImage mode **77**
links
  changing the behavior **90**
  computing the endpoints **90**
  computing the shape **90**
  creating **102**
  creating custom links **96**
  creating polyline links **103**
  description **87**, **89**
  editing **104**
  end **89**
  how they are drawn **90**
  intermediate points **96**
  lightweight **91**
  managing **87**
  oriented mode **89**
  predefined classes **91**
loading
  prototype library **130**

M

makeColumn member function
  IlvManager class **31**
makeLink member function
  IlvGrapher class **87**
  IlvMakePolyLinkInteractor class **103**

MakeNode command **75**
makeNode member function
  IlvGrapher class **87**
makeRow member function
  IlvManager class **31**
manager grid **64**
manager view hooks
  description **61**
  example **62**
managers
  and views **18**
  applying functions in a region **28**
  binding views **20**
  commands **19**
  displaying groups and instances **155**
  double-buffering **20**, **22**
  hooks **20**
  input/output **20**
  modifying geometric properties of objects **27**
  optimizing drawing tasks **33**
  overview **16**
  reading **34**
  saving **34**
  selecting objects **28**, **29**
  selection procedures **29**
  zooming **22**
manual
  naming conventions **13**
  notation **13**
  typographic conventions **13**
mapLinks member function
  IlvGrapher class **88**
menu bar **138**
miscellaneous accessors
  description **203**
modifying object states
  and managers **68**
moveObject member function
  IlvManager class **28**
movePoints member function
  IlvPolylineLinkImage class **96**
multiple views
  and managers **20**
  description **18**

N

naming conflicts **183**
naming conventions **13**
NewGrapherBuffer command **74**, **79**, **137**
NewGraphicBuffer command **74**, **137**
NewProtoLibrary command **145**
NewPrototype command **145**
NewPrototypeEditionBuffer command **145**
NewPrototypeGrapherBuffer command **145**
nodes
  arranging **88**
  connecting **102**
  creating **102**
  description **87**
  managing **87**
  retrieving links **88**
  testing connection **88**
nodeXPretty member function
  IlvGrapher class **88**
nodeYPretty member function
  IlvGrapher class **88**
notation **13**
numberOfSelections member function
  IlvManager class **29**

O

object interactors
  and managers **52**
  description **52**
object properties
  and managers **30**
objects
  managing **27**
OneLinkImage mode **77**
OneSplineLinkImage mode **77**
OpenProtoLibrary command **146**
orientation **93**
OrientedArcLinkImage mode **77**
OrientedDoubleLinkImage mode **78**
OrientedDoubleSplineLinkImage mode **78**
OrientedLinkImage mode **78**
OrientedOneLinkImage mode **78**
OrientedOneSplineLinkImage mode **78**

OrientedPolylineLinkImage mode **78**

P

palettes panel **140**
parameters
- direct **154**
- input **154**
- object/node **154**
- output **154**

perpendicular lines **92**
pin editor mode **75**
PolylineLinkImage mode **78**
prototype accessors
- Assign **184**
- Blink **197**
- Callback **200**
- Clock **201**
- Condition **185**
- Debug **203**
- Event **202**
- Fill **192**
- Format **186**
- Group **181**
- Increment **186**
- Invert **198**
- Min/Max **187**
- Multiple **188**
- MultiRep **192**
- Notify **189**
- Prototype **204**
- Reference **181**
- Rotate **199**
- Rotation **193**
- ScaleX **194**
- ScaleY **195**
- Script **182**, **189**
- Toggle **191**
- TranslateX **195**
- TranslateY **196**
- Value **180**
- Watch **201**

prototype library
- creating **115**
- loading **130**
- saving **130**

Prototype Studio
- buffer types **131**
- connecting prototype instances **132**
- creating
  - prototype instances **131**
- creating prototype library **115**
- creating prototypes **115**
- defining attributes of a prototype **116**
- drawing a prototype **119**
- editing
  - panels with prototype instances **131**
  - prototype instances **132**
  - prototype nodes **120**
- extending **175**
- loading
  - prototype libraries **130**
  - prototype panels **132**
- saving
  - prototype libraries **130**
  - prototype panels **132**
  - prototypes **129**
- structuring prototype nodes **121**

prototypes
- accessor definition **152**
- accessor parameters **123**, **154**
- advantages **111**
- architecture **150**
- compiling applications **158**
- connecting instances **132**
- creating **110**
  - by coding **173**
  - instances **131**, **160**
  - with IBM ILOG Views Studio **110**
- creating prototype library **115**
- deleting instances **160**
- design pattern **108**
- design pattern definition **111**
- drawing graphic elements **119**
- editing
  - instances **132**
- examples **109**
- extending **175**
- getting attributes **161**
- group mediators **166**

groups **151**
header files **158**
instances **131**, **155**
librairies **155**
librairies for compiling **158**
linking
application objects **156**, **165**
loading
prototype instances **158**
overview **108**
proto mediators **168**
retrieving instances **161**
saving **129**
setting attributes **161**
setting values directly **165**
specifying
graphical behavior **112**
interactive behavior **112**
structuring nodes **121**
sub-attributes **165**
using in applications **110**, **157**
values **152**
Prototypes buffer window
description of **138**
prototypes extension **140**

R

read method
IlvContainer class **158**
IlvManager class **158**
readFile method
IlvContainer class **158**
IlvManager class **158**
readObject member function
IlvGraphInputFile class **88**
IlvManagerInputFile class **35**
reDo member function
IlvManager class **66**
reDraw member function
IlvManager class **32**
reDrawViews member function
IlvManager class **33**
removeAccelerator member function
IlvManager class **53**
removeEventHook member function
IlvManager class **37**
removeInteractor member function
IlvManager class **37**
removePoints member function
IlvPolylineLinkImage class **96**
removeView member function
IlvManager class **21**
removeViewHook member function
IlvManager class **61**
reshapeObject member function
IlvManager class **28**
rotateView member function
IlvManager class **22**

S

sameHeight member function
IlvManager class **31**
sameWidth member function
IlvManager class **31**
SaveProtoLibraryAs command **146**
saving
prototype library **130**
prototypes **129**
segment layout
automatic **94**
fixed **94**
SelecArcLinkImageMode command **79**
SelectDoubleLinkImageMode command **80**
SelectDoubleSplineLinkImageMode command **80**
SelectGroupConnectionMode command **146**
SelectGroupSelectionMode command **147**
selecting
objects **29**
selection procedures
and managers **29**
example **29**
SelectLinkImageMode command **80**
SelectNodeSelectionMode command **147**
SelectOneLinkImageMode command **81**
SelectOneSplineLinkImageMode command **81**
SelectOrientedArcLinkImageMode command **81**
SelectOrientedDoubleLinkImageMode command **81**

SelectOrientedDoubleSplineLinkImageMode command **82**
SelectOrientedLinkImageMode command **82**
SelectOrientedOneLinkImageMode command **82**
SelectOrientedOneSplineLinkImageMode command **83**
SelectOrientedPolylineLinkImageMode command **83**
SelectPinEditorMode command **83**
SelectPolylineLinkImageMode command **83**
setAlpha member function
  IlvManagerLayer class **26**
setBackground member function
  IlvManager class **23**
setControlPoint member function
  IlvOneSplineLinkImage class **93**
setDoubleBuffering member function
  IlvManager class **22**
setFactory member function
  IlvMakeLinkInteractor class **103**
  IlvMakeNodeInteractor class **102**
setFixedOffset member function
  IlvArcLinkImage class **95**
setFixedOrientation member function
  IlvDoubleLinkImage class **94**
setInteractor member function
  IlvManager class **37**
setLabel member function
  IlvLinkLabel class **92**
setMakeSelection member function
  IlvManager class **29**
setModified member function
  IlvManager class **68**
setNumLayer member function
  IlvManager class **24**
setOffsetRatio member function
  IlvArcLinkImage class **95**
setOrientation member function
  IlvOneLinkImage class **93**
setOriented member function
  IlvLinkImage class **89**
setPinIndex member function
  IlvGrapherPin class **98**
setPoints member function
  IlvPolylineLinkImage class **96**
setSelected member function
  IlvManager class **28**
setTo member function
  IlvLinkImage class **89**
setTransformer member function
  IlvManager class **22**
setUndoEnabled member function
  IlvManager class **66**
shortCut member function
  IlvManager class **41**, **53**
ShowApplicationInspector command **142**
ShowGroupEditor command **147**
smooth curves **94**
snapping grids **64**
snapToGrid member function
  IlvManager class **65**
start node **89**

T

three connected lines **94**
ToggleTimers command **147**
toolbar
  editing modes **140**
transformerChanged member function
  IlvManagerViewHook class **61**
translateObject member function
  IlvManager class **28**
translateView member function
  IlvManager class **22**
trigger accessors
  description **200**
typographic conventions **13**

U

unDo member function
  IlvManager class **66**
  IlvTranslateObjectCommand class **67**
undo/redo actions **66**
unGroup member function
  IlvManager class **31**
UngroupIlvGroups command **148**
update region **33**

V

view hooks **60**
view interactors
  and managers **37**
  extending **45**
  manager example **39**
  predefined in managers **38**
`viewRemoved` member function
  `IlvManagerViewHook` class **61**
`viewResized` member function
  `IlvManagerViewHook` class **61**
views
  adding **21**
  and managers **18**
  getting **21**
  multiple **18**, **20**
  removing **21**

W

windows
  2D Graphics **74**, **137**
  Grapher **74**, **137**
  Prototypes **138**
`writeObject` member function
  `IlvGraphOutputFile` class **88**
  `IlvManagerOutputFile` class **35**

Z

`zoomView` member function
  `IlvManager` class **22**