IBM ILOG Views Manager V5.3 User's Manual

June 2009

Table of Contents

Preface **About This Manual 5**

What You Need to Know 5

Manual Organization 5

Notation. 6

Typographic Conventions 6

Naming Conventions 6

Chapter 1 **Basic Manager Features 7**

Introducing Managers 7

Layers 8

Views 9

View Transformer. 9

Event Handling. 9

Main Features of IlvManager 10

Manager Views 11

View Transformations 13

Double-buffering. 13

Manager Layers 14

Layer Index 15

Layer Selectability 16
Layer Visibility 16
Layer Rendering. 17
Managing Objects 18
Modifying the Geometry of Graphic Objects 18
Selecting Objects 19
Selection Procedures 20
Managing Selected Objects 20
Managing Object Properties. 21
Arranging Objects 21
Drawing and Redrawing 23
Optimizing Drawing Tasks 24
Saving and Reading. 25

Chapter 2 Manager Event Handling 27

The Event Handling Mechanism. 27
Event Hooks 28
View Interactors 28
Predefined View Interactors 29
Example: Implementing the IlvDragRectangleInteractor Class 30
Example of an Extension: IlvMoveInteractor 36
Object Interactors 43
Accelerators 43
Example: Changing the Key Assigned to an Accelerator. 44
Predefined Manager Accelerators 44

Chapter 3 Advanced Manager Features. 47

Observers 47
General Notifications 48
Manager View Notifications 48
Manager Layer Notifications 49
Manager Contents Notifications. 50

Graphic Object Geometry Notifications 50
Example 50
View Hooks 51
Manager View Hooks 52
Example: Monitoring the Number of Objects in a Manager 53
Example: Maintaining a Scale Displayed With No Transformation 53
Manager Grid 55
Example: Using a Grid 56
Undoing and Redoing Actions 57
Command Class. 57
Managing Undo 57
Example: Using the IlvManagerCommand Class to Undo/Redo 58
Managing Modifications 59

Index 61

About This Manual

This *User's Manual* 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.

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 the following chapter:

- **Chapter 1** describes the main principles behind using managers.
- **Chapter 2** describes how events are handled when using managers.
- **Chapter 3** describes advanced features of managers.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in `courier` typeface.
- Entries to be made by the user are written in *`courier italics`*.
- Some words in *italics*, when seen for the first time, may be found in the glossary at the end of this manual.

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 Foundation library begin with `Ilv`.
- The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

```
class IlvDrawingView;
```

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

```
virtual IlvClassInfo* getClassInfo() const;

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

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:

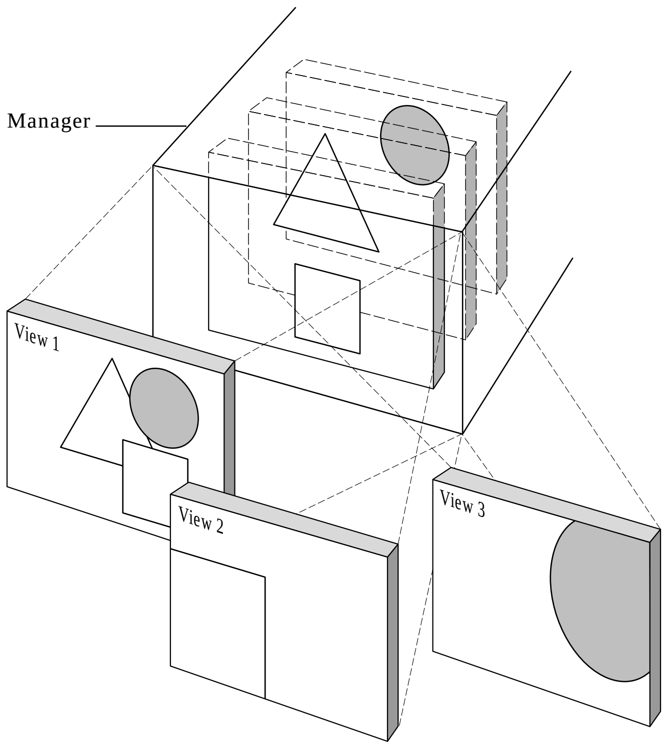

Figure 1.1 *Manager Concept*

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.

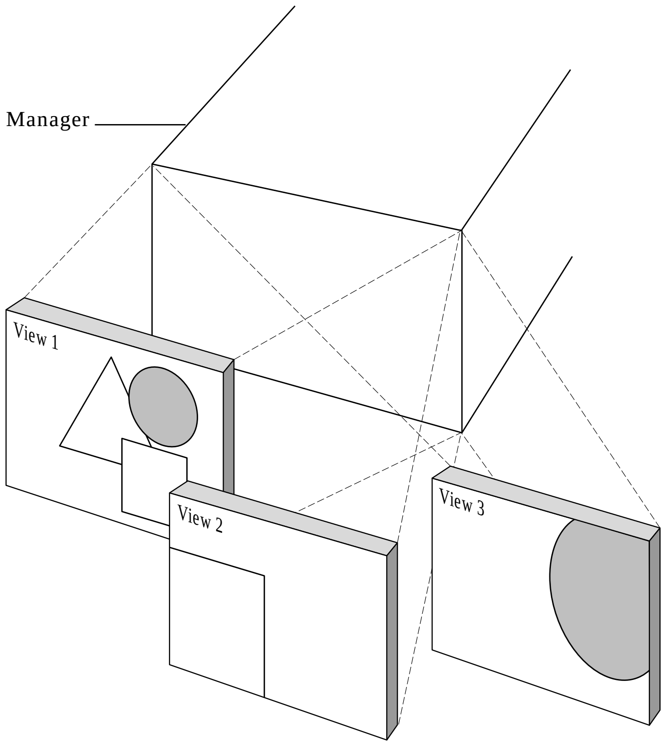

Figure 1.2 Multiple Views Bound To a Manager

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.

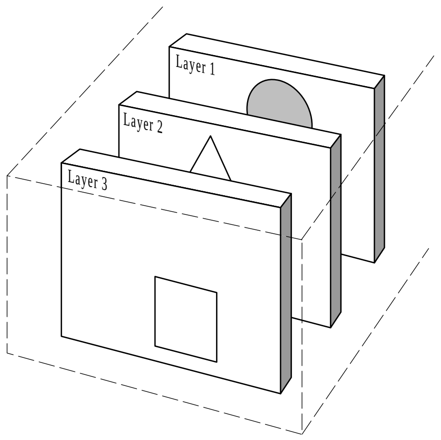

Figure 1.3 *Layers*

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`.

C H A P T E R

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.

C H A P T E R

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);
```

Index

A

abortReDraws member function
  IlvManager class **24**
accelerators
  and managers **44**, **48**
  example in managers **44**
  predefined in managers **44**
addAccelerator member function
  IlvManager class **44**
addCommand member function
  IlvManager class **57**
addTransformer member function
  IlvManager class **13**
addView member function
  IlvManager class **12**
addVisibilityFilter member function
  IlvManagerLayer class **17**
afterDraw member function
  IlvManagerViewHook class **52**
afterExpose member function
  IlvManagerViewHook class **52**
align member function
  IlvManager class **22**
applyInside member function
  IlvManager class **19**
applyIntersects member function
  IlvManager class **18**, **19**
applyToInside member function
  IlvManager class **18**
applyToObject member function
  IlvManager class **18**, **19**, **25**
applyToObjects member function
  IlvManager class **18**
applyToSelections member function
  IlvManager class **18**, **41**
applyToTaggedObjects member function
  IlvManager class **18**

B

beforeDraw member function
  IlvManagerViewHook class **52**
binding views **11**
bufferedDraw member function
  IlvManager class **23**

C

C++
  prerequisites **5**
commands
  and managers **11**
contentsChanged member function
  IlvManager class **59**
  IlvManagerViewHook class **52**
copy member function
  IlvTranslateObjectCommand class **59**

D

deleteSelections member function
  IlvManager class **20**
deSelectAll member function
  IlvManager class **20**
displaying objects
  and managers **21**
  drawing **23**
doIt member function
  IlvDragRectangleInteractor class **34**
  IlvMakeRectangleInteractor class **35**
  IlvMoveInteractor class **40**
  IlvTranslateObjectCommand class **58**
double-buffering
  and managers **13**
  description **11**
draw member function
  IlvManager class **23**
drawGhost member function
  IlvDragRectangleInteractor class **34**
  IlvMoveInteractor class **39**
duplicate member function
  IlvManager class **23**

E

ensureVisible member function
  IlvManager class **13**
events
  and accelerators **10**
  and interactors **9**
  and managers **9**

F

fitToContents member function
  IlvManager class **13**
fitTransformerToContents member function
  IlvManager class **13**, **44**
forgetUndo member function
  IlvManager class **57**

G

geometric transformations
  and managers **9**
  and views **9**
getAccelerator member function
  IlvManager class **44**
getInteractor member function
  IlvManager class **28**
getSelections member function
  IlvManager class **20**
getViews member function
  IlvManager class **12**
global functions
  IlvGetContentsChangedUpdate **59**
  IlvSetContentsChangedUpdate **59**
graphic objects
  and managers **19**
  selecting in manager **19**
grids
  and managers **55**
  example **56**
  snapping **55**
group member function
  IlvManager class **22**
grouping
  and managers **22**

H

handleEvent member function
  IlvDragRectangleInteractor class **31**
  IlvMoveInteractor class **37**
hooks **11**, **51**

I

IlvContainer class **10**
IlvDragRectangleInteractor class **29**
  doIt member function **34**
  drawGhost member function **34**
  handleEvent member function **31**
IlvEllipse class **29**
IlvFilledRectangle class **29**
IlvFilledRoundRectangle class **29**

IlvGadgetManagerInputFile class **26**
IlvGadgetManagerOutputFile class **26**
IlvGetContentsChangedUpdate global function **59**
IlvGraphic class
  description **8**
  scale member function **19**
  translate member function **19**
IlvGraphicSet class **22**
IlvInputFile class **25**
IlvInteractor class **9**
IlvLayerVisibilityFilter class
  isVisible member function **17**
IlvLine class **20**
IlvLineHandle class **20**
IlvMakeArrowInteractor class **30**
IlvMakeBitmapInteractor class **30**
IlvMakeEllipseInteractor class **29**
IlvMakeFilledEllipse class **30**
IlvMakeFilledEllipseInteractor class **29**
IlvMakeFilledRectangleInteractor class **29**
IlvMakeFilledRoundRectangleInteractor class **29**
IlvMakeLineInteractor class **30**
IlvMakeRectangleInteractor class **29**
  description **29**
  doIt member function **35**
IlvMakeReliefDiamondInteractor class **29**
IlvMakeReliefLineInteractor class **30**
IlvMakeReliefRectangleInteractor class **29**
IlvMakeRoundRectangleInteractor class **29**
IlvMakeUnZoomInteractor class **30**
IlvMakeZoomInteractor class **30**
IlvManager class
  abortReDraws member function **24**
  addAccelerator member function **44**
  addCommand member function **57**
  addTransformer member function **13**
  addView member function **12**
  align member function **22**
  applyInside member function **18**, **19**
  applyIntersects member function **18**, **19**
  applyToObject member function **18**, **19**, **25**
  applyToObjects member function **18**
  applyToSelections member function **18**, **41**
  applyToTaggedObjects member function **18**
  bufferedDraw member function **23**
  contentsChanged member function **59**
  deleteSelections member function **20**
  deSelectAll member function **20**
  draw member function **23**
  duplicate member function **23**
  ensureVisible member function **13**
  fitToContents member function **13**
  fitTransformerToContents member function **13**, **44**
  forgetUndo member function **57**
  getAccelerator member function **44**
  getInteractor member function **28**
  getSelection member function **20**
  getSelections member function **20**
  getViews member function **12**
  group member function **22**
  initReDraws member function **24**
  installEventHook member function **28**
  installViewHook member function **52**
  invalidateRegion member function **24**
  isDoubleBuffering member function **13**
  isInvalidating member function **24**
  isModified member function **59**
  isSelected member function **19**
  isUndoEnabled member function **57**
  makeColumn member function **22**
  makeRow member function **22**
  moveObject member function **19**
  numberOfSelections member function **20**
  reDo member function **57**
  reDraw member function **23**
  reDrawViews member function **24**
  removeAccelerator member function **44**
  removeEventHook member function **28**
  removeInteractor member function **28**
  removeView member function **12**
  removeViewHook member function **52**
  reshapeObject member function **19**
  rotateView member function **13**
  sameHeight member function **22**
  sameWidth member function **22**
  setBackground member function **14**
  setDoubleBuffering member function **13**
  setInteractor member function **28**

setMakeSelection member function **20**
setModified member function **59**
setNumLayer member function **15**
setSelected member function **19**
setTransformer member function **13**
setUndoEnabled member function **57**
shortCut member function **32**, **44**
snapToGrid member function **56**
translateObject member function **19**
translateView member function **13**
unDo member function **57**
unGroup member function **22**
view transformation member functions **13**
zoomView member function **13**
IlvManagerclass
aligning objects member functions **22**
main features **10**
IlvManagerCommand class
advanced features **57**
example **58**
instances **10**
IlvManagerEventHook class **28**
IlvManagerGrid class **55**
IlvManagerInputFile class
description **25**
readObject member function **26**
IlvManagerLayer class
addVisibilityFilter member function **17**
setAlpha member function **17**
IlvManagerObjectInteractor class **43**
IlvManagerOutputFile class
description **25**
example **26**
writeObject member function **26**
IlvManagerViewHook class
afterDraw member function **52**
afterExpose member function **52**
beforeDraw member function **52**
contentsChanged member function **52**
description **52**
interactorChanged member function **52**
transformerChanged member function **52**
viewRemoved member function **52**
viewResized member function **52**
IlvManagerViewInteractor class **9**, **28**, **36**
IlvMoveInteractor class
doIt member function **40**
drawGhost member function **39**
example **36**
handleEvent member function **37**
IlvOutputFile class
description **25**
IlvRectangle class **35**
IlvReliefDiamond class **29**
IlvRoundRectangle class **29**
IlvSelectInteractor class **30**
IlvSetContentsChangedUpdate global function **59**
IlvTextField class **53**
IlvTranslateObjectCommand class
copy member function **59**
description **58**
doIt member function **58**
unDo member function **58**
initReDraws member function
IlvManager class **24**
installEventHook member function
IlvManager class **28**
installViewHook member function
IlvManager class **52**
interactorChanged member function
IlvManagerViewHook class **52**
interactors
view **28**
invalidateRegion member function
IlvManager class **24**
isDoubleBuffering member function
IlvManager class **13**
isInvalidating member function
IlvManager class **24**
isModified member function
IlvManager class **59**
isSelected member function
IlvManager class **19**
isUndoEnabled member function
IlvManager class **57**
isVisible member function
IlvLayerVisibility class **17**

L

layers
    and managers **9**, **14**
    default number **15**
    description **8**
    object selectability **16**
    object visibility **16**
    setting up **15**

M

makeColumn member function
    IlvManager class **22**
makeRow member function
    IlvManager class **22**
manager grid **55**
manager view hooks
    description **52**
    example **53**
managers
    and views **9**
    applying functions in a region **19**
    binding views **11**
    commands **10**
    double-buffering **11**, **13**
    hooks **11**
    input/output **11**
    modifying geometric properties of objects **18**
    optimizing drawing tasks **24**
    overview **7**
    reading **25**
    saving **25**
    selecting objects **19**, **20**
    selection procedures **20**
    zooming **13**
manual
    naming conventions **6**
    notation **6**
    organization **5**
modifying object states
    and managers **59**
moveObject member function
    IlvManager class **19**
multiple views
    and managers **11**
    description **9**

N

naming conventions **6**
notation **6**
numberOfSelections member function
    IlvManager class **20**

O

object interactors
    and managers **43**
    description **43**
object properties
    and managers **21**
objects
    managing **18**

R

readObject member function
    IlvManagerInputFile class **26**
reDo member function
    IlvManager class **57**
reDraw member function
    IlvManager class **23**
reDrawViews member function
    IlvManager class **24**
removeAccelerator member function
    IlvManager class **44**
removeEventHook member function
    IlvManager class **28**
removeInteractor member function
    IlvManager class **28**
removeView member function
    IlvManager class **12**
removeViewHook member function
    IlvManager class **52**
reshapeObject member function
    IlvManager class **19**
rotateView member function
    IlvManager class **13**

S

`sameHeight` member function
  `IlvManager` class **22**
`sameWidth` member function
  `IlvManager` class **22**
selecting
  objects **20**
selection procedures
  and managers **20**
  example **20**
`setAlpha` member function
  `IlvManagerLayer` class **17**
`setBackground` member function
  `IlvManager` class **14**
`setDoubleBuffering` member function
  `IlvManager` class **13**
`setInteractor` member function
  `IlvManager` class **28**
`setMakeSelection` member function
  `IlvManager` class **20**
`setModified` member function
  `IlvManager` class **59**
`setNumLayer` member function
  `IlvManager` class **15**
`setSelected` member function
  `IlvManager` class **19**
`setTransformer` member function
  `IlvManager` class **13**
`setUndoEnabled` member function
  `IlvManager` class **57**
`shortCut` member function
  `IlvManager` class **32**, **44**
snapping grids **55**
`snapToGrid` member function
  `IlvManager` class **56**

T

`transformerChanged` member function
  `IlvManagerViewHook` class **52**
`translateObject` member function
  `IlvManager` class **19**
`translateView` member function
  `IlvManager` class **13**

U

`unDo` member function
  `IlvManager` class **57**
  `IlvTranslateObjectCommand` class **58**
undo/redo actions **57**
`unGroup` member function
  `IlvManager` class **22**
update region **24**

V

view hooks **51**
view interactors
  and managers **28**
  extending **36**
  manager example **30**
  predefined in managers **29**
`viewRemoved` member function
  `IlvManagerViewHook` class **52**
`viewResized` member function
  `IlvManagerViewHook` class **52**
views
  adding **12**
  and managers **9**
  getting **12**
  multiple **9**, **11**
  removing **12**

W

`writeObject` member function
  `IlvManagerOutputFile` class **26**

Z

`zoomView` member function
  `IlvManager` class **13**