IBM ILOG Views

Prototypes V5.3

Tutorial

June 2009

Copyright notice

Trademarks

Notices

Table of Contents

About This Tutorial

This tutorial introduces you to creating and using business graphic objects with the Prototypes extension of IBM® ILOG® Views Studio.

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 chapters:

- *Creating Prototypes in IBM ILOG Views Studio*
- *Linking Your Prototype to Application Objects*

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

```
class IlvDrawingView;
```

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

```
virtual IlvClassInfo* getClassInfo() const;

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

C H A P T E R

1

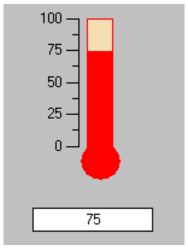
Creating Prototypes in IBM ILOG Views Studio

This section provides a short tutorial on creating and using prototypes. The tutorial takes approximately 20 minutes to complete and includes the following sections:

- *Creating a Prototype*.
- *Defining the Application Interface of the Prototype*.
- *Defining the Visual Representation of the Prototype*.
- *Defining the Graphic Behavior of the Prototype*.
- *Creating an Instance of a Prototype*.
- *Defining an Interactive Behavior for a Prototype*.
- *Connecting Two Prototype Instances*.

Introduction

In this tutorial, you are going to create the graphical interface of the following object, called temperature. The temperature object resembles a thermometer, with a scale and a gauge that indicates the level of the temperature. When the temperature exceeds 30, the gauge appears in red. When the temperature is below 30, the gauge appears in blue. Under the thermometer, a text field displays the temperature in numerals.



The temperature object has two attributes: the temperature attribute and the threshold attribute. The user should have visual feedback of whether the temperature is above or below the threshold. You will define the application interface of the prototype by describing these two attributes. Then you will create its graphical representation using the graphical editor of IBM ILOG Views Studio . Finally, you will create the graphical and interactive behaviors, that is, how the attribute values are shown to the user and how the user can edit the temperature attribute.

The final step of application integration, connecting a prototype instance to a real application written in C++ and providing data, is described in the next tutorial.

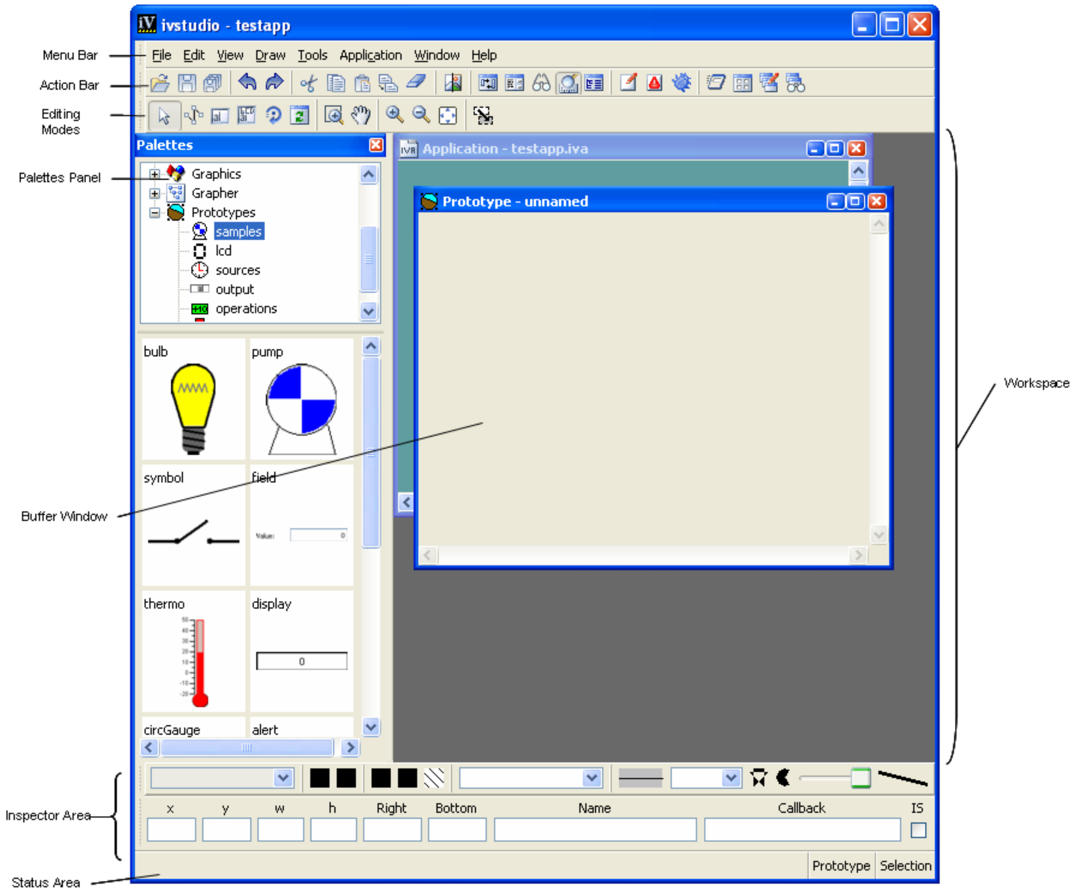
Launching Studio With the Prototypes Extension

To launch Studio with the Prototypes extension:

1. Go to the directory `$ILVHOME/studio/<system>` of the IBM ILOG Views distribution.

2. Type `ivfstudio` or double-click the Studio icon to start the application.

   The Studio Main window appears on your screen. When first opened, the Main window displays an empty 2D Graphics buffer window. The Palettes panel contains the predefined Prototypes libraries, as well as the other IBM ILOG Views Studio predefined libraries.



Creating a Prototype

You will create your prototype in a Prototype buffer window. Since you want to store the prototype for later reuse from a library, you must first create the library to store it.

1. To create a new prototype library, choose New from the File menu. Then choose Prototype Library from the submenu that is displayed.

   The Save As dialog box appears.

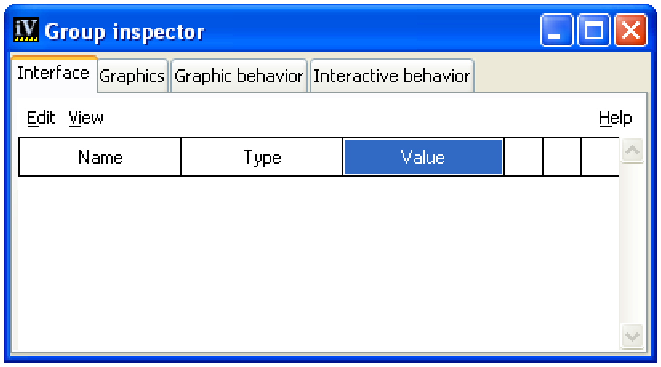
2. Choose a directory where you have write permission. Type the file name `myLib.ipl` and click Save.

   A new tree item, called `myLib`, appears below the predefined Prototypes libraries in the upper pane of the Palettes panel. You have created your prototype library.

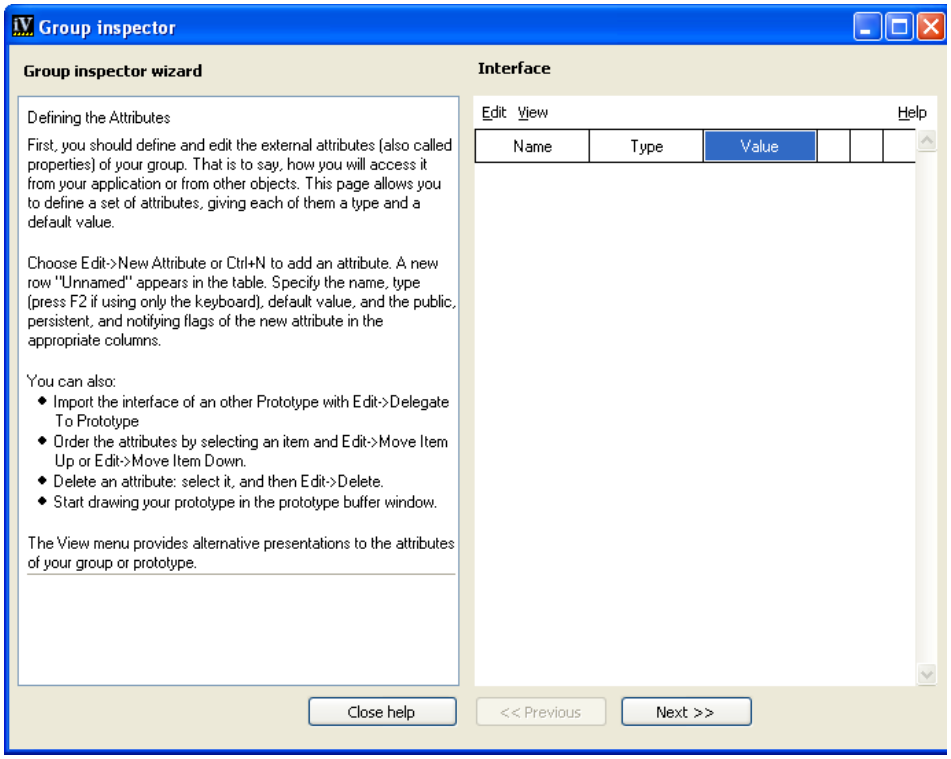
3. To open a new Prototype buffer window, choose New from the File menu. Then choose Prototype from the submenu that is displayed.

A Prototype buffer window, called “Prototype - unnamed”, is displayed in the Main window.

The Group Inspector panel also appears:



By clicking the Help button, a contextual hypertext page opens, guiding you through the process of creating your prototype:



Defining the Attributes

First, you should define and edit the external attributes (also called properties) of your group. That is to say, how you will access it from your application or from other objects. This page allows you to define a set of attributes, giving each of them a type and a default value.

Choose Edit->New Attribute or Ctrl+N to add an attribute. A new row "Unnamed" appears in the table. Specify the name, type (press F2 if using only the keyboard), default value, and the public, persistent, and notifying flags of the new attribute in the appropriate columns.

You can also:

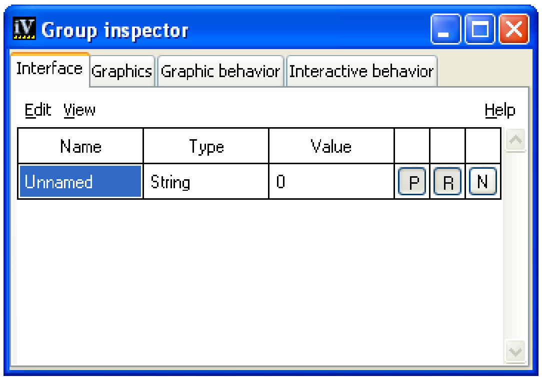
- Import the interface of an other Prototype with Edit->Delegate To Prototype
- Order the attributes by selecting an item and Edit->Move Item Up or Edit->Move Item Down.
- Delete an attribute: select it, and then Edit->Delete.
- Start drawing your prototype in the prototype buffer window.

The View menu provides alternative presentations to the attributes of your group or prototype.

Defining the Application Interface of the Prototype

The application interface is the set of attributes that you can set and retrieve from the application. These attributes determine the external appearance of the prototype. You define these by adding attributes in the Interface page of the Group Inspector panel.

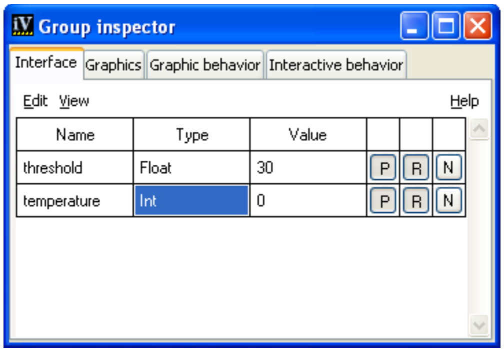
1. Choose Edit > New Attribute. A new attribute “Unnamed” appears.



Name	Type	Value			
Unnamed	String	0	P	R	N

2. Click `Unnamed` and then type `threshold`, thereby renaming the attribute from `Unnamed` to `threshold`.
3. In the Type column, click `String` twice, or press F2. A combo box appears, letting you choose the type of the attribute. Select Value > Float to specify that your attribute will be of type Float.
4. To set a default value of 30 in the Value column, click the `0` in the `threshold` attribute row and type `30`.

   Leave the default settings in the last three columns: P is depressed indicating that your attribute will be public; R is depressed indicating that your attribute will be persistent; N is not depressed indicating that your attribute will not notify others of its changes.
5. To create another attribute named `temperature`, select Edit > New Attribute. Rename `Unnamed` to `temperature`.
6. Set its type to `Int` by opening the combo box in the Type column and selecting Value > Int.
7. Finally, this `temperature` attribute should notify others of its changes. Therefore, click the N button in the last column to set it.



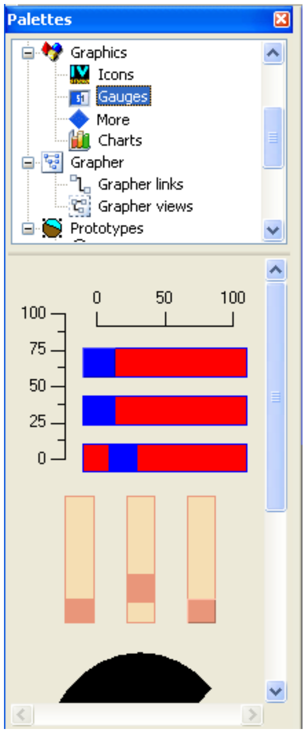
Name	Type	Value			
threshold	Float	30	P	R	N
temperature	Int	0	P	R	N

***Note:** It is important to define the threshold attribute before the temperature attribute, because the behavior of* `temperature` *depends on the value of* `threshold`.

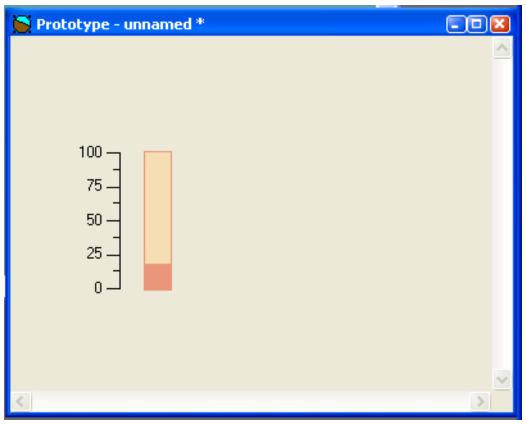
Defining the Visual Representation of the Prototype

You will create the visual representation of your prototype.

1. First, select the next tab of the Group Inspector, to display the Graphics page.
2. Select the Prototype buffer window by clicking in “Prototype - unnamed” in the Main window of IBM ILOG Views Studio.
3. In the upper pane of the Palettes panel, use the scroll bars on the right side of the screen to move down through the list of palettes until the Graphics item appears. Choose Gauges under the Graphics item to display the palette containing gauges and scales.



4. Drag a vertical rectangular scale (`IlvRectangularScale`) from the palette to the "Prototype - unnamed" window. Then drag a vertical rectangular gauge (`IlvRectangularGauge`) and place it next to the scale. Stretch the gauge vertically so that it matches the height of the scale:



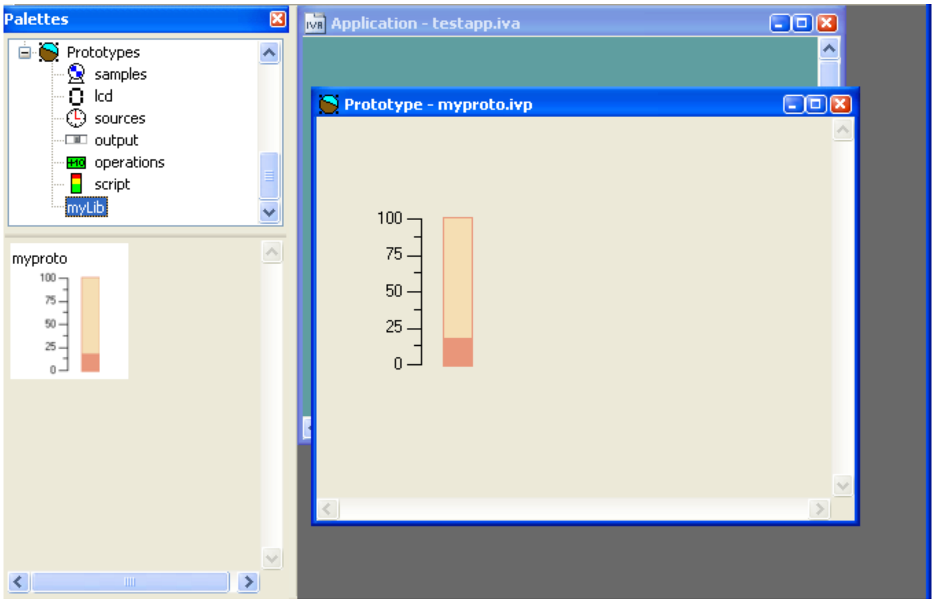
5. Choose Save As from the File menu to save the prototype to the prototype library.

6. A Message dialog box asks you if you want to save the prototype in a library. Click Yes.

7. A Prompt String dialog box appears to let you choose the library in which to save your new prototype. Make sure `myLib` is selected in the list of libraries, or type `myLib` in the text field. Click Apply.

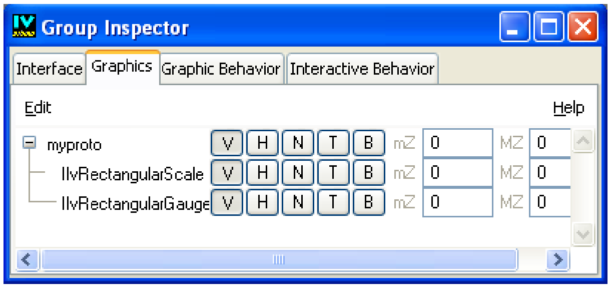
8. A new Prompt String dialog box asks you to name the prototype. Type `myproto` and click Apply.

In the Main window, the Prototype buffer window name changes to `myproto.ivp` and a new prototype icon, labeled `myproto`, appears in the Prototypes palette on the `myLib` page.



9. If necessary, bring the Graphics panel of the Group Inspector to the front by choosing Group Inspector from the Tools menu of the Main window and clicking the Graphics tab.

A hierarchical sheet showing the prototype structure and the attributes of each node appears on the page:



myproto	V	H	N	T	B	mZ	0	MZ	0
IlvRectangularScale	V	H	N	T	B	mZ	0	MZ	0
IlvRectangularGauge	V	H	N	T	B	mZ	0	MZ	0

10. Select the graphic node `IlvRectangularScale`. You may want to enlarge the window to better view the names of the objects, or move the pointer over a name to see the tooltip giving the full name of the object underneath the pointer.

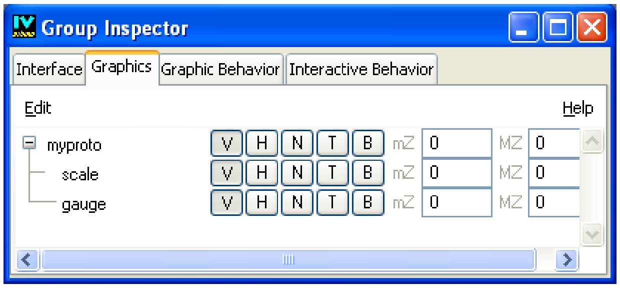
Notice that the scale is selected in the Main window as well.

11. Click again to make the field editable, and type `scale`. Press the Enter key.

The `IlvRectangularScale` in the tree gadget changes to `scale`.

12. Repeat steps 8 and 9 to rename the `IlvRectangularGauge` node as `gauge`.

The Graphics notebook page of the Group Inspector should look like this:



myproto	V	H	N	T	B	mZ	0	MZ	0
scale	V	H	N	T	B	mZ	0	MZ	0
gauge	V	H	N	T	B	mZ	0	MZ	0

13. Choose Save from the File menu of the Main window to save the prototype.

Defining the Graphic Behavior of the Prototype

Once you have defined the interface and the graphic objects that make up your prototypes, you can start linking them together by adding behaviors. These determine how a graphic object will change when an attribute of the prototype changes.

1. Move to the Graphic Behavior tab of the Group Inspector.

2. Select the `temperature` attribute in the attributes tree.

3. Select Control > Assign.

   A new behavior is added. You now have to specify the parameters of this behavior.

4. Click twice in the right hand column of the `Attribute` parameter of the new assigned behavior to open the combo box.

5. Select the item `gauge` > `value`, or type `gauge.value` in the text field and press Enter.

6. Click twice in the right hand column of the `Send` parameter of the assigned behavior to open the combo box.

7. Choose [All Types] to display all the parameters and select the item `temperature`, or type `temperature` and press Enter.

This assigned behavior specifies that when the temperature is to be changed, it will set the value attribute of the gauge to the current value of the temperature.

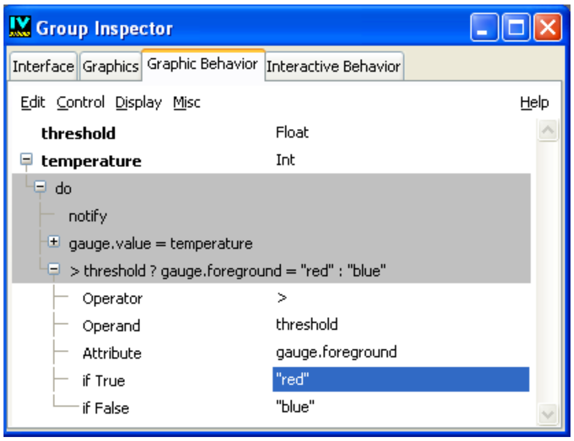
You are going to add a second action to the `temperature` accessor. This action will specify that when the temperature rises above the threshold value, the gauge will appear red and when the temperature is below the threshold value, the gauge will appear blue.

1. If not still selected, select the `temperature` attribute.

2. Select Control > Condition. This adds a new behavior below the previous assigned behavior.

3. There are 5 parameters to set for this behavior. Using the same techniques as before, clicking twice in the right hand column and selecting from the resultant combo box, set the following values for each parameter:

   - Operator: >
   - Operand: [All Types] > threshold
   - Attribute: gauge > foreground
   - if True: [Immediate value], then type `red` in the text field. (You can also select the item Choose... in the combo box and choose a color using the color selector.)
   - if False: [Immediate value], then type `blue` in the field to enter the color.



threshold	Float
temperature	Int
do	
notify	
gauge.value = temperature	
> threshold ? gauge.foreground = "red" : "blue"	
Operator	>
Operand	threshold
Attribute	gauge.foreground
if True	"red"
if False	"blue"

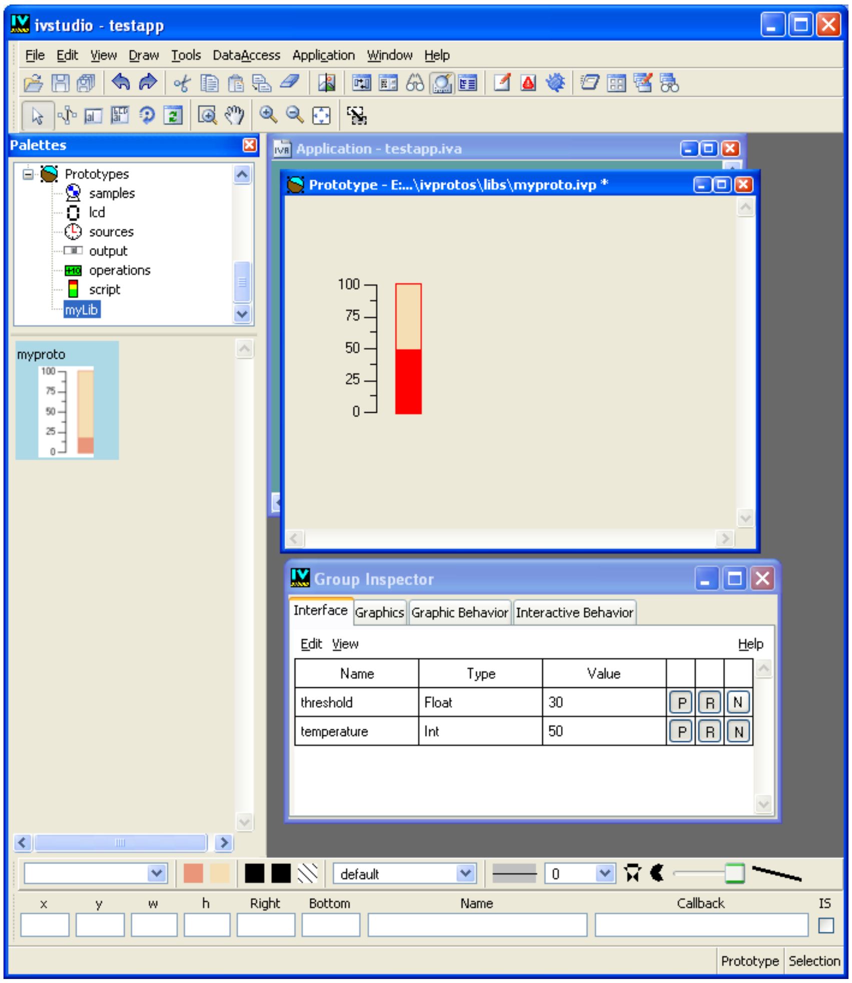
Changing the value of `temperature` is associated with two actions. First, the actual level of the gauge changes. Second, the condition is tested and the color of the gauge is set accordingly.

Changing a Prototype Value

Next, you are going to change the value of the `temperature` accessor to test the behavior of your prototype.

1. Click the Interface tab in the Group Inspector panel to bring the notebook page to the foreground. The list of attributes appears showing a `temperature` value of `0`.
2. Click in the value column and change `0` to `50`. Press the Enter key.

   In the Prototype buffer window, you will notice that the level of the gauge changes to show a value of 50 and the gauge turns to red.



Name	Type	Value			
threshold	Float	30	P	R	N
temperature	Int	50	P	R	N

3. Change the value from 50 to 20 and press the Enter key.

   You will notice that the gauge shows a value of 20 and turns to blue.

4. Choose Save from the File menu to save your latest changes.

You have completed creating the first version of your prototype. Next, you will learn how to instantiate a prototype in an Application buffer window.

Creating an Instance of a Prototype

Once a prototype has been created, it can be instantiated in an application buffer window. The application will have access to its attributes when it loads the panel or it can dynamically add instances representing application objects.

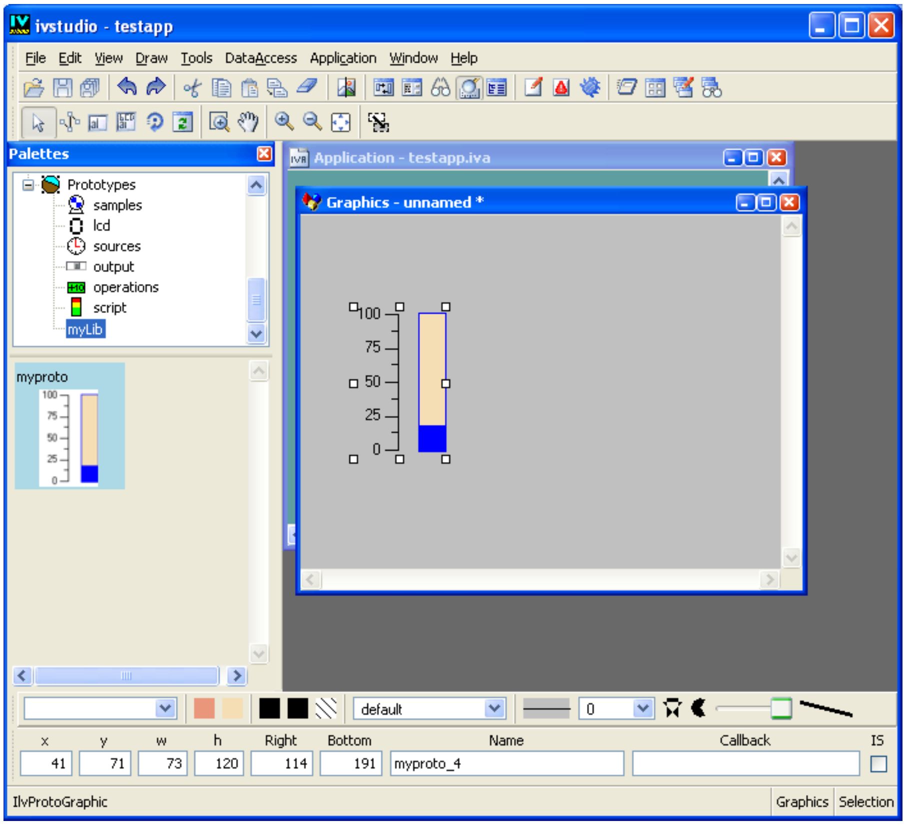
You are going to create an instance of your prototype in an application buffer window.

1. To open a buffer window, choose New from the File menu and then 2D Graphics from the submenu that is displayed.

   The buffer window is opened in the Main window. This buffer window can hold prototypes or other graphic objects.

2. Drag the `myproto` prototype icon from the `myLib` Prototypes palette to the Main window.

   The prototype instance is created and is selected.



x	y	w	h	Right	Bottom	Name	Callback	IS
41	71	73	120	114	191	myproto_4		☐

3. Go to the Group Inspector panel.

   Notice that only the Interface and Graphics notebook pages are active. The other notebook pages are not active, so you cannot add behaviors or change the nodes of a prototype instance. You can, however, change the temperature and threshold values. You can also change some global properties of the nodes, such as the capability to zoom in and out. (The instance must remain selected in the Main window.) You can also create several instances of your prototype and give them different values.

   In order to access your prototype instance, you will most likely want to give a meaningful name to the instance you have just created.

4. If not still selected, select the instance.
5. In the name field of the Graphics page of the Group Inspector, type `myThermometer`.

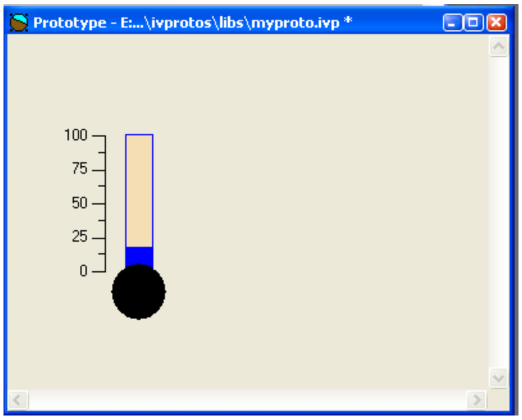
   You will be able to retrieve it in your C++ program with the method `getGroup("myThermometer")` of the `IlvGroupHolder` class.

6. Choose Save As from the File menu and save this window with the file name `myPanel.ilv`.

Modifying the Prototype

You are going to modify your prototype to make it look more like a thermometer.

1. Choose `myproto` from the Window menu in the Main window to bring the Prototype buffer window to the foreground.
2. In the upper pane of the Palettes panel, use the scroll bar to move down the tree gadget and click `Graphics`.
3. From the Graphics palette displayed in the lower pane, drag a filled ellipse (`IlvFilledEllipse`) over to the `myproto.ivp` window.
4. Reshape and resize the ellipse. Place it under the gauge so that the combination of the two graphic objects looks like a thermometer:



5. Click on the Graphic Behavior tab in the Group Inspector panel to bring the corresponding notebook page to the foreground.

6. Ensure that no attributes are currently selected. If necessary, choose Edit > Deselect, or deselect the selected item by clicking on it using the middle mouse button.

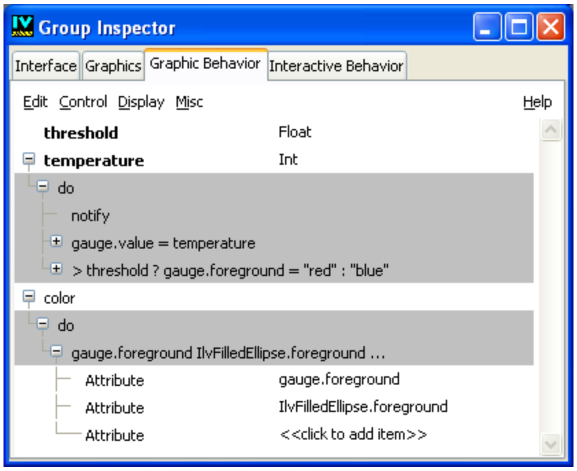
7. Select Control > Multiple. This has the effect of creating a new intermediate attribute, named “Action”.

8. Click on the `Action` label and type `color` to set the name of this attribute. Press Enter.

9. In the right hand column of the first `Attribute` parameter, click twice and choose [All Types]. Select gauge > foreground from the combo box menu. Alternatively you can directly type `gauge.foreground` and press Enter.

10. Click twice on the second parameter, labeled “<<click to add item>>”. Choose [All Types], and then select `IlvFilledEllipse` > `foreground` from the combo box menu.

You have added an intermediate attribute, `color`, that will address both the gauge and the ellipse foreground when set.



```
threshold                                  Float
⊟ temperature                              Int
  ⊟ do
      notify
    ⊞ gauge.value = temperature
    ⊞ > threshold ? gauge.foreground = "red" : "blue"
⊟ color
  ⊟ do
    ⊟ gauge.foreground IlvFilledEllipse.foreground ...
        Attribute                          gauge.foreground
        Attribute                          IlvFilledEllipse.foreground
        Attribute                          <<click to add item>>
```

You need to edit the `temperature.Condition` accessor to use the `color` accessor instead of the `gauge.foreground` attribute.

1. In the behaviors tree, expand the `temperature` > `Condition` accessor by clicking the [+] button at the left of the line `> threshold ? gauge.foreground =...`

   The `gauge.foreground` parameter appears in the matrix on the right side of the page, ready to be edited.

2. Click twice on that parameter, open the combo box and select `color`.
3. This replaces the former definition of the accessor with the new one which changes the foreground color of both the ellipse and the gauge when the temperature is modified.
4. Choose Save from the File menu of the Main window to save your prototype.

You can move back to the Interface page if you wish, and test the new behavior of the prototype by changing the temperature value and seeing that both the ellipse and the gauge are affected by the condition behavior.

Finally, simplify the temperature attributes. Because this temperature will always be equal to the internal `gauge.value` attribute, you can change its type. Instead of being a simple `Int` value, you can make the temperature attribute directly reference the `gauge.value`. To do this:

1. Go to the Graphic Behavior page of the Group Inspector.
2. In the right hand side of the `temperature` attribute, click twice on the `Int` label that indicates its type. A combo box appears.
3. Choose Reference > gauge > value in the combo box menu.

The `Int` type is replaced by `^gauge.value` indicating that changing the temperature is equivalent to directly setting the `gauge.value` subattribute.

4. Select the `gauge.value=temperature` line in the tree. Then choose Edit > Delete.

   This has the effect of removing the redundant assignment.

Defining an Interactive Behavior for a Prototype

You are going to define an interactive behavior for your prototype. This is done by adding Event behaviors, defined in the Interactive Behavior page, the remaining page of the Group Inspector.

1. Choose `myproto` from the Window menu in the Main window to bring the Prototype buffer window to the foreground.
2. Click the Interactive Behavior tab in the Group Inspector panel.
3. Choose Events > Callback in the menu.

   A new attribute named “Action” is created, as well as a Callback behavior and a “Watch” behavior.

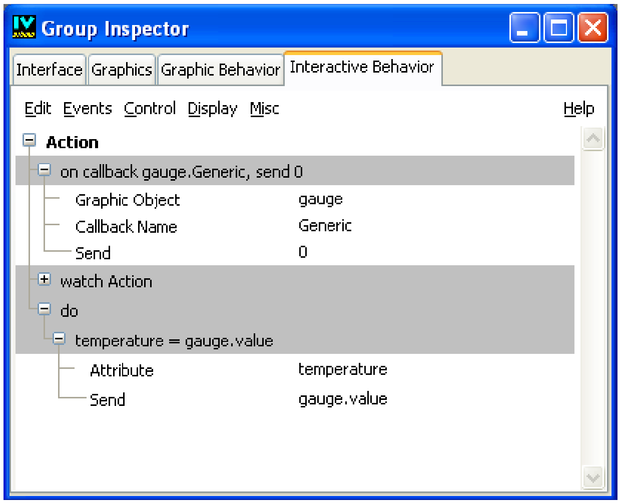
4. In the matrix on the right side of the page, set the Graphic Object parameter to `gauge`. Leave the Callback Name and Input parameters on their default settings of `Generic` and `0` respectively.

   The Callback accessor specifies that when the Generic callback of the gauge is called (through user input), the temperature value will be queried and sent to whatever object listens to the changes of the `Action` value. These “listeners” can be other accessors of the same prototype, other prototypes, or application objects. By default, adding the Callback behavior has added to itself a “Watch” behavior.

5. Choose Control > Assign from the menu bar.

   A new assigned behavior is added at the bottom of the `Action` behavior.

6. In the matrix on the right side, select the parameter values as follow:
   - Attribute: `temperature`
   - Send: `gauge` > `value`



```
Action
  on callback gauge.Generic, send 0
    Graphic Object        gauge
    Callback Name         Generic
    Send                  0
  watch Action
  do
    temperature = gauge.value
      Attribute           temperature
      Send                gauge.value
```

This means that when the callback is triggered through user action, the temperature attribute is adjusted to match the gauge value.

7. Choose Save from the File menu of the Main window to save the prototype.

You can test the interactive behavior of your prototype:

1. In the Prototype buffer window, select the Active mode, either by issuing a marking menu command (right, then left), or by choosing the Active icon in the Editing Modes toolbar.

2. Click within the gauge object, or drag the level up or down. You will see the gauge update itself, as well as change color when you cross the threshold value. In the Group Inspector Interface page, you will see the temperature update itself each time you end the interaction.

Editing a Prototype

Once you have created and saved a prototype, you can edit it again by selecting the prototype in the palette, and choosing View > Edit Group (Ctrl+E).

All the changes you make will be propagated to the instances you have created when you save it.

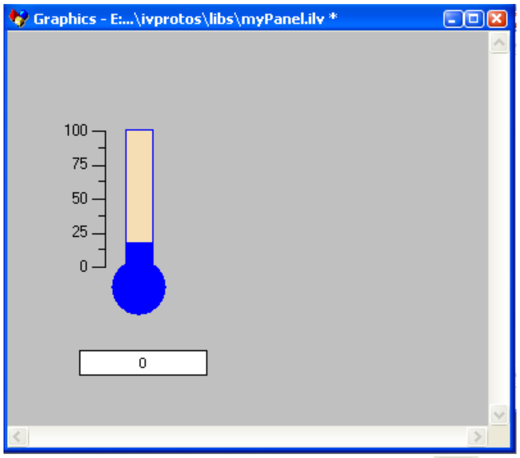
For instance, you can select the “bulb” prototype in the “samples” palette, and double click on its icon: the Prototype edition buffer is opened, and the group inspector allows you to edit its interface and behaviors.

Connecting Two Prototype Instances

You will learn how to connect two different prototype instances.

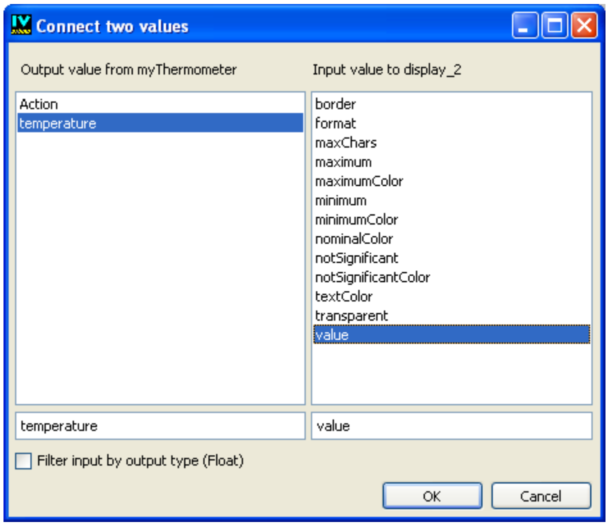
1. Choose `myPanel` from the Window menu in the Main window.
2. The 2D Graphics buffer window should contain the prototype instance that you created in the tutorial. If it does not, drag the `myproto` icon from the Prototypes palette to the buffer window to create a new instance of the prototype.
3. In the Palettes panel, choose `samples` in the Prototypes palette.
4. Drag the display icon to the buffer window.

   This creates an instance of the `display` prototype in the same panel as the `myproto` prototype instance:



5. Select the Group Connection interactor from the Editing Modes toolbar.
6. Click in the `myproto` prototype instance. Drag a line from the `myproto` instance to the `display` instance. When you start dragging, be sure that you are dragging from a graphic object and not from an empty space.

   When you release the mouse button, the Connect two values dialog box appears asking you to specify which output value from the `myproto` instance you wish to connect to which input value of the `display` prototype:



Output value from myThermometer

Input value to display_2

Left pane: Action, temperature (selected)

Right pane: border, format, maxChars, maximum, maximumColor, minimum, minimumColor, nominalColor, notSignificant, notSignificantColor, textColor, transparent, value (selected)

temperature | value

☐ Filter input by output type (Float)

7. In `myproto`, the triggering value is `temperature`. Select `temperature` in the left pane and `value` in the right pane. Click the OK button.

   A green arrow appears going from the `myproto` instance to the `display` instance, indicating that a connection is made.

   Each time the temperature is changed, either by the user or the application, the `value` accessor of the `display` instance will be notified and will change its value accordingly.

8. To test this action, switch to Active mode by clicking the Active icon in the Editing Modes toolbar.

   Change the level of the gauge by dragging it with the mouse. The label in the `display` instance changes as you change the level of the gauge.

You have completed the first prototypes tutorial.

Summary

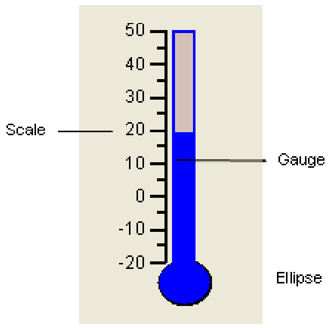
The prototype that you have created in this tutorial is a simplified version of the `thermo` prototype, one of the predefined prototypes of the sample library. The `thermo` prototype (and also the `myproto` prototype that you created) has three components:

- **Application interface**: this is composed of its public values and defines how the application is to manipulate the prototype.

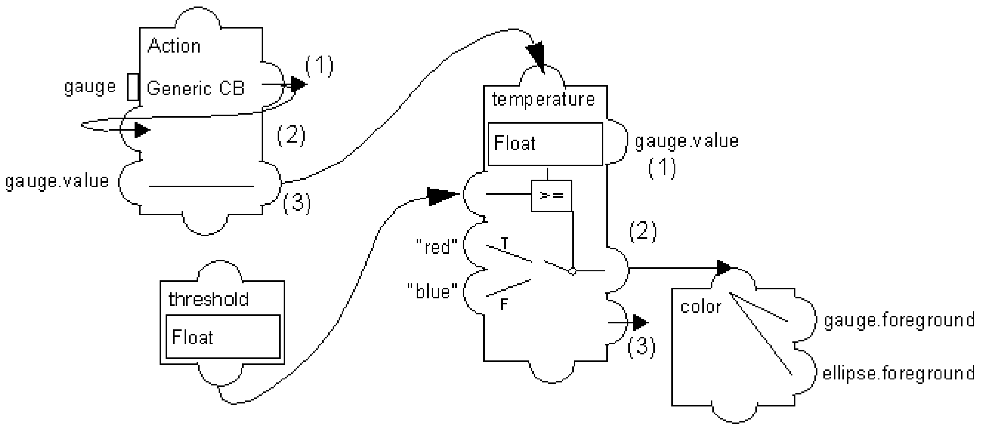
  `Threshold`: Float, the alert temperature you want to draw the user's attention to.

`Temperature`: `Int`, the main value.

◆ **Graphical representation**: this is how the user perceives the application interface values. It is composed of three objects: a scale, a gauge, and an ellipse:



◆ **Behavior (Graphic and Interactive)**: this is represented using the following dataflow graph showing the relationships between accessors:



This dataflow graph is meant to provide an overview of the dependencies among the accessors that define the prototype behavior. At this point, you do not need to understand the conventions used in the graph to represent accessors.

The behavior of the prototype is defined by the following four attributes:

- `threshold` - holds a float value that is used when setting the `temperature` value. It is important that this attribute be defined before `temperature`, as `temperature` is dependent on it. When the prototype is read back, `threshold` is set to its value before `temperature` is evaluated, so that the condition accessor tests a correct value when it is evaluated for the first time. An alternate way to ensure that the behaviors are executed in a workable order would be to add an Assigned behavior temperature=temperature to the threshold behavior, to force a reevaluation of temperature when threshold is modified.

- temperature - holds the following behaviors:

  (1) A Reference accessor: it references the gauge.value internal attribute. gauge.value can also be called a subattribute of the prototype.

  (2) A Condition accessor: when the temperature is above the threshold, the value "red" is sent to the color attribute. Otherwise, "blue" is sent.

  (3) A Notify accessor: this indicates that when the value is changed, this attribute can notify others of its changes.

- color - is a Multiple accessor that sends its value to the Ellipse and Gauge foreground colors.
- Action - holds the following accessors:

  (1) A Callback accessor: whenever the user triggers the Generic callback of the gauge (by clicking and dragging), a value is sent to whatever other accessor needs to be kept aware of changes in the gauge value.

  (2) A Watch accessor: it indicates that it wants to be kept aware of changes to the Action attribute (which is itself).

  (3) An Assign accessor: it indicates that whenever the Action attribute is changed, the gauge.value subattribute should be assigned to the temperature value. This is slightly redundant given that temperature is a reference to gauge.value itself, but is necessary to interpret the other behaviors of the temperature attribute (condition and notify).

Finally, you may want to connect a prototype instance to a real application providing data. This is the topic of the next tutorial.

You may also want to examine in more detail the resources of the behaviors mechanism, and how to create more complex behaviors. The samples libraries that come with IBM ILOG Views Studio are a good starting point to further examine the various effects that can be created with prototypes.

Predefined Libraries

You can look at the predefined prototypes contained in the IBM ILOG Views distribution. Prototype Studio loads these libraries at start-up time.

Library Name	Description
samples	Sample library loaded at start-up.
sources	Prototypes containing value sources.

Library Name	Description
`output`	Prototypes containing gadgets and defining output values.
`lcd`	LCD displays (one digit and four digits).
`operations`	Prototypes that can be used to connect prototypes and execute operations on their values.
`script`	Prototypes that use script accessors.

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

You can look at any prototype definition by double-clicking the prototype icon. This will load the prototype into a Prototype buffer window and open a Group Inspector panel. When you load a panel file that contains prototype instances, the required prototype libraries are automatically loaded in the Prototypes palette.

Samples closer to real applications are available in the directory `$ILVHOME/samples/protos` of the distribution. Each of these samples comes packaged in a subdirectory with an example program that uses a prototype library (usually held in the data subsubdirectory of the sample).

See the index.html file in the `$ILVHOME/samples/protos` in the library for a complete description.

C H A P T E R

2

Linking Your Prototype to Application Objects

This tutorial introduces the procedures available for linking prototypes to application objects. It takes approximately 40 minutes to complete. You will learn how to perform the following basic tasks:

- ◆ Load a panel containing Prototype instances.
- ◆ Retrieve the instances you want to drive from your application data.
- ◆ Create a Group Mediator to handle user events and feed them into your application.
- ◆ Link the Group Mediator to your application objects.

Introduction

This tutorial is organized into three steps, as follows:

- ◆ *Step 1: Reading a Panel Containing Prototype Instances*
- ◆ *Step 2: Retrieving and Modifying the Prototype Instances*
- ◆ *Step 3: Creating a Group Mediator to Handle User Feedback*

To start the tutorial, copy the files located in `$ILVHOME/doc/tutorials/protos/tutorial2` into a directory where you have write permission. These files form the basic template from which you can proceed through this tutorial. For each step X, you will find files in the stepX directory that correspond to this step and build instructions that depend on your environment.

First, start IBM ILOG Views Studio with the Prototypes extension and open the file `myPanel.ilv` to examine its contents: it holds a prototype instance similar to the one built in the first tutorial (see *Creating Prototypes in IBM ILOG Views Studio*). The prototype for this instance is held in the `myLib.ipl` file, which opens automatically when you open the file.

Step 1: Reading a Panel Containing Prototype Instances

The file `step1/program.cpp` contains the basic canvas of an IBM® ILOG® Views program, with which you should already be familiar.

In order to have the prototypes library working, it is necessary to include the following files, specific to the prototypes modules:

```
#include <ilviews/protos/proto.h>
#include <ilviews/protos/protogr.h>
#include <ilviews/protos/groupholder.h>
#include <ilviews/protos/allaccs.h>
```

***Note:** Other includes would be needed to use other features of IBM ILOG Views, such as Managers.*

While not mandatory, it is often convenient to include all default graphic objects in the main file. This enables you to edit a prototype in IBM ILOG Views Studio without having to ensure that all the individual graphic objects have been included. Once you have finished modifying or extending your prototypes, you may wish to include only the necessary objects in order to optimize the runtime footprint.

```
#include <ilviews/graphics/all.h>
```

For this particular example, you will also need to include the following file in order to read the display prototype that uses the `IlvMessageLabel` class:

```
#include <ilviews/gadgets/msglabel.h>
```

If you have IBM ILOG Views Controls, including all predefined gadgets will let you use the gadgets in your prototypes:

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

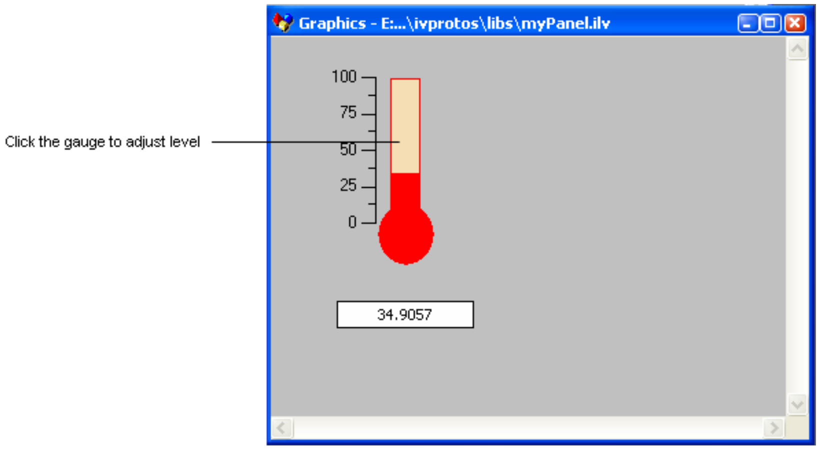
To run a program that contains calls to the Prototype package, you have to link the program with (at least) these libraries: `ilvproto`, `ilvgrapher`, `ilvmgr`, `views`, `ilog`, and with the specific libraries for your platform. Only the library `ilvproto` is specific to the Prototype package.

***Note:** The makefiles and project files provided have already specified all the libraries needed to compile, link, and execute this program.*

The program opens a display. You then add ".." to the display path to locate the prototype library `myLib.ipl` which holds the prototypes already defined. Next, the program creates a container and loads the file `myPanel.ilv`, containing prototype instances. These instances are automatically loaded. Finally, the program shows the container and starts the main loop, like all regular IBM ILOG Views programs:

```
int
main(int argc, char* argv[])
{
  // Connect to the display system.
  IlvDisplay* display = new IlvDisplay("ViewsBGO", 0, argc, argv);
  if (!display || display->isBad()) {
    IlvFatalError("Couldn't open display");
    return -1;
  }
  // Prepare a window and a manager to display a scene.
  IlvManager* manager = new IlvManager(display);
  IlvView* view = new IlvView(display, "BGO tutorial 1",
        "BGO tutorial 1", IlvRect(0, 0, 400, 300));
  manager->addView(view);
  // Load a data file into the manager
  // Add the samples data directory to the display path
  // to be sure to find the proper file.
  IlString buf (getenv("ILVHOME"));
  buf+="/doc/tutorial2/step1";
  display->appendToPath(buf);
  display->appendToPath("..");
  manager->read("myPanel.ilv");
  // The prototype instances are automatically read.
  // Then run the main loop.
  manager->setDoubleBuffering(view, IlvTrue);
  view->show();
  IlvMainLoop();
  return 0;
}
```

You can compile and link this program (`step1/program.cpp`) to see that your panel has been read, which will operate as indicated here:


Graphics - E:...\ivprotos\libs\myPanel.ilv
Click the gauge to adjust level
100
75
50
25
0
34.9057

In the next section you will add some code to retrieve the prototype instance `myThermometer`, which you created in the panel in the first tutorial (see *Creating Prototypes in IBM ILOG Views Studio*).

Step 2: Retrieving and Modifying the Prototype Instances

All operations done on prototype instances within a panel, be it a container or a manager, (and on `IlvGroups`, which are the superclass of prototype instances) are done using an `IlvGroupHolder`. This class allows you to add, remove, and retrieve objects of class `IlvGroup` from a panel.

After having read the file, retrieve the group holder associated with the manager. From the group holder, retrieve the prototype instance `myThermometer`, and initialize its temperature attribute to `0.0`:

```
// Insert the following code after the line:
// "the prototype instances are automatically read."
  IlvGroupHolder* holder=IlvGroupHolder::Get(manager);
  IlvGroup* myThermometer = holder->getGroup("myThermometer");
  if (!myThermometer) {
    IlvFatalError("This program expects to find an IlvGroup "
              "of name 'myThermometer' in the file 'myPanel.ilv'");
    return -1;
  }
  myThermometer->changeValue(IlvValue("temperature", 0.0));
```

Next, you need to periodically update the temperature value from values provided by the application. For this, create a timer routine by inserting the following lines before the main function in the program (after the include directives):

```
// Insert the following code before the main() body.
const IlUInt angleincrement = 4;
```

```
static void
TimerProc(IlvTimer*, IlvAny arg)
{
static IlUInt angle=0;
    IlvGroup* thermometer=(IlvGroup*) arg;
    IlDouble temperature=50.0+40.0*sin(degreesToRadians(angle));
    // Feed in temperature values.
    thermometer->changeValue(IlvValue("temperature", temperature));
    angle = (angle + angleincrement) % 360;
}
```

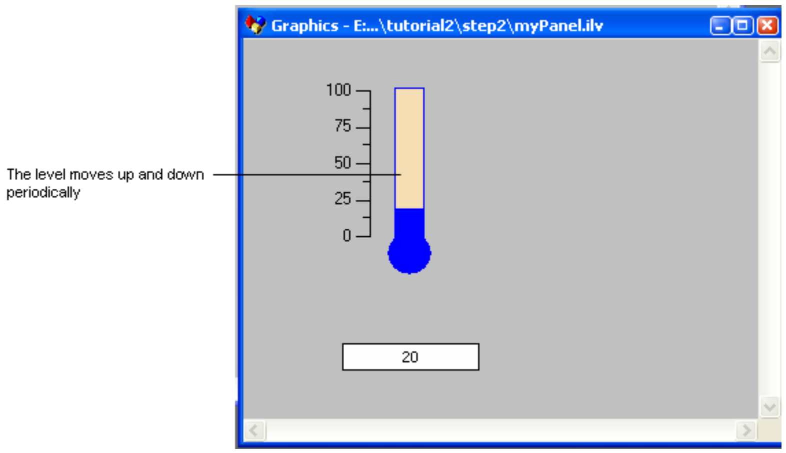
This function makes the temperature rise and fall according to a sinusoidal function, between the values of 10 and 90. Next, initialize this timer to wake up every 200 ms and update the prototype instance "`myThermometer`", which you retrieved from the panel. After the line:

```
   myThermometer->changeValue(IlvValue("temperature", 0.0));
```

insert the following code:

```
  // start changing the values of the target group
  IlvTimer* timer = new IlvTimer(display, 0, 200, TimerProc,
                                  (IlvAny)myThermometer);
  timer->run();
```

This will periodically update the thermometer values. You can now compile and run your program. You will see that the temperature rises and falls regularly, the thermometer becoming red when the temperature is above the threshold previously set in IBM ILOG Views Studio.


Graphics - E:...\tutorial2\step2\myPanel.ilv
100
75
50
25
0
20
The level moves up and down periodically

You can change the threshold value with a call to:

```
myThermometer->changeValue(IlvValue("threshold", 50.0));
```

You can retrieve the threshold value with the following call:

```
IlvValue tval("threshold", (IlDouble)0.0);
IlDouble threshold = (IlDouble) myThermometer->queryValue(tval);
```

The complete source code is provided in the file `step2/program.cpp`, which serves as a starting point for the next step of the tutorial.

Step 3: Creating a Group Mediator to Handle User Feedback

You want the application not only to drive the display, but also to handle some user input that affects internal parameters. To do this, the prototype `myproto` in IBM® ILOG® Views Studio has been slightly extended to add a slider-like object. The user will use this slider to change the animation rate of the thermometer, making the temperature rise and fall faster or slower depending on its value.

You may want to open the library `myLib.ipl` in IBM ILOG Views Studio to check the attributes and behaviors that have been added in order to achieve this effect.

Linking an application object and a prototype instance is done using the `Mediator` design pattern, implemented by the class `IlvGroupMediator`. To use this class, you should include the following file:

```
#include <ilviews/protos/grouplin.h>
```

Defining a Token Application Object

In this tutorial, assume that there is one simple object, a thermometer, that you want to display and control through a user interface implemented with BGOs:

```
#if defined(ILVSTD)
#include <cmath> // for sin() function
ILVSTDUSE
#else
#include <math.h>
#endif
struct Thermometer {
    IlFloat temperature;
    IlUInt acceleration;
    IlUInt curval;
    IlvTimer timer;
// The display argument is here to allow the use of a Views timer.
    Thermometer(IlvDisplay*);
    ~Thermometer() {}
    void adjust_temp();
static void TimerProc(IlvTimer*, IlAny);
};

Thermometer::Thermometer(IlvDisplay* dpy)
:temperature(20), acceleration(4), curval(0),
 timer(dpy, 0, 200, TimerProc, this)      //  see Note that follows
{
     timer.run();
}

void Thermometer::adjust_temp()
{
   temperature = 50.0 + (40.0 *
                  sin(degreesToRadians((IlDouble)curval)));
   curval = (curval + acceleration) % 360;
}
void Thermometer::TimerProc(IlvTimer*, IlAny arg)
{
   ((Thermometer*)arg)->adjust_temp();
}
```

Note: *Some systems may raise a warning about the use of "this" in the construction above. The construction is safe, however, and the warning may be ignored.*

The method `adjust_temp` is called periodically to implement the dynamics of the application. You will notice that this class does not offer any user-interface specific functionality. It is meant to function as a stand-alone piece of software that can work with other application objects (in a plant simulator, for instance), but nothing has been done to allow the interactive viewing or editing of its values. In this tutorial, you will create a user interface for this object without modifying its definition at all.

In the main program, after creating the display, create an instance of this application object:

```
// create and initialize an application object.
Thermometer myThermometer(display);
```

Defining the GroupMediator of an Application's Class

To create a user interface for an instance of `Thermometer`, you define a subclass of `GroupMediator` that handles this `Thermometer` class:

```
struct TemperatureWatcher: public IlvGroupMediator
{
    TemperatureWatcher(IlvGroup* g, Thermometer* a)
    : IlvGroupMediator(g, a) {
        if(!tempSymbol) tempSymbol=IlvGetSymbol("temperature");
        if(!accSymbol) accSymbol=IlvGetSymbol("acceleration");
    }
    IlBoolean changeValues(const IlvValue*, IlvUShort);
    void queryValues(IlvValue*, IlvUShort) const;
inline Thermometer* getThermometer() const {
            return (Thermometer*) getObject(); }
static IlvSymbol* tempSymbol;
static IlvSymbol* accSymbol;
};
IlvSymbol* TemperatureWatcher::tempSymbol;
IlvSymbol* TemperatureWatcher::accSymbol;
```

You need to define how to update the values of `Thermometer` when they are edited by the user. The `changeValues` method establishes a one-to-one correspondence between attribute values of the prototype instances and the `Thermometer` members:

```
// Method handling with user input and updating the application.
IlBoolean
TemperatureWatcher::changeValues(const IlvValue* v, IlvUShort n)
{
    if(locked()) return IlFalse;
    for (IlUInt i=0;i<n;i++) {
        if (v[i].getName() == accSymbol)
           getThermometer()->acceleration = (IlUInt) v[i];
// You do not want the temperature value to be editable by the user.
// If this was the case, the following code would simply be added:
//         else if (v[i].getName() == tempSymbol)
//           getThermometer()->temperature = (IlFloat) v[i];
    }
    return IlTrue;
}
```

You need to specify through which prototype attributes the values of Thermometer are to be reflected in the user interface. The `queryValues` method establishes here a one-to-one correspondence between the `Thermometer` attributes and the values of a prototype instance representing it:

```
// Method that synchronizes the group to the application values.
void TemperatureWatcher::queryValues(IlvValue* v, IlvUShort n) const
{
    for (IlUInt i=0;i<n;i++) {
        if (v[i].getName() == tempSymbol)
            v[i] = getThermometer()->temperature;
        else if (v[i].getName() == accSymbol)
            v[i] = getThermometer()->acceleration;
    }
}
```

Linking an Application Object to its Representation

To instantiate the user interface for the Thermometer application, create an instance of `myThermometer` that watches it. In the main function, add the following code after having retrieved the group `myThermometer` from the group holder (but before showing the main window):

```
    TemperatureWatcher watch(tempRep, &myThermometer);
    watch.update();
    IlvTimer timer(display, 0, 200, TimerProc, &watch);
    timer.run();
```

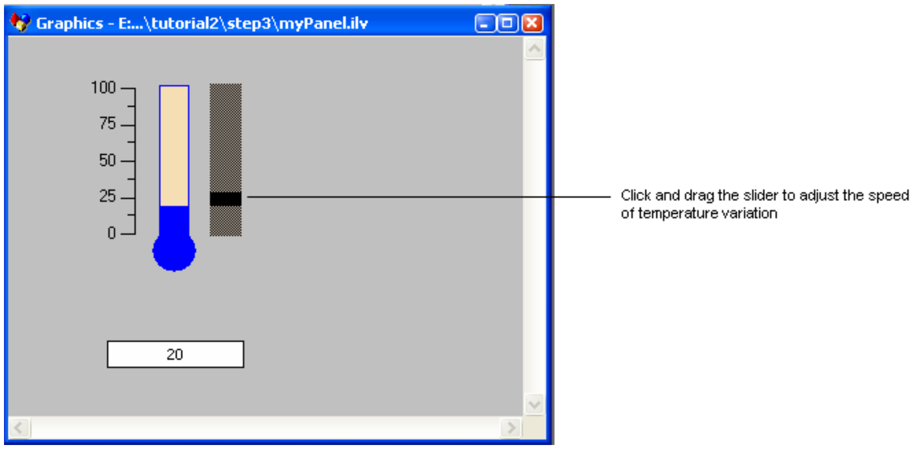
The timer will periodically check the state of the thermometer instance to update the presentation if needed.

Synchronizing the Application Object and its Display

Finally, you need to define the asynchronous update routine before the main function:

```
static void
TimerProc(IlvTimer*, IlvAny arg)
{
    TemperatureWatcher* watcher=(TemperatureWatcher*) arg;
    watcher->update();
}
```

The `TemperatureWatcher` will periodically update itself according to the current values of the `Thermometer` objects.


Graphics - E:...\tutorial2\step3\myPanel.ilv
100
75
50
25
0
20
Click and drag the slider to adjust the speed of temperature variation

Note: *Other mechanisms could be implemented, such as having the Thermometer class behave according to the Observer/Observable design pattern and being able to notify other objects of its changes. This method is slightly more efficient. The samples in* `$ILVHOME/samples/protos` *implement this technique. However, the use of a triggered asynchronous observer is a useful technique to implement applications that have real-time constraints: at any time you can adjust the rate at which the user interface is refreshed, if needed to meet severe time constraints.*

Summary

You can now compile and run the program to test its behavior. If you wish, you can compile and run the file `step3/program.cpp`. This file implements the tutorial as described, and also adds a few extra hints and probable optimizations that you may want to perform. You will notice that the temperature rises and falls as in the program at the end of step 2. In addition however, you can adjust the acceleration rate with the slider-like object to the right of the thermometer.

The mechanism described enables the easy implementation of user-interfaces while preserving strong encapsulation of application objects. It becomes possible with this design to completely separate the implementation of the functional core from the user interface and have them evolve separately, once a common interface has been agreed upon and group mediators have been defined. In the `samples/protos` directory several other mechanisms to integrate BGOs in your application are shown to let you find the one that best suits your needs.

Index