



IBM ILOG Views
Application Framework V5.3
User's Manual

June 2009

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

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

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

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

Trademarks

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

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

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

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

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

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

Notices

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

Table of Contents

Preface	About This Manual	7
	What You Need to Know	7
	Manual Organization	7
	Notation	8
	Typographic Conventions	8
	Naming Conventions	8
Chapter 1	Introducing IBM ILOG Views Application Framework	9
	What is Application Framework	9
	The Document/View Architecture	10
Chapter 2	Using the Application Framework Editor	13
	Starting Up the Application Framework Editor	13
	Application Framework Editor Main Window	14
	Components Palette	15
	Workspace	16
	Creating a New Application	17
	Selecting a Document Type	18
	Creating and Configuring an Options File (.odv file)	19
	Setting Application Parameters	19

	Adding Menu Items	20
	Adding Toolbar Items	21
	Setting Document Parameters	21
	Setting General Document Parameters	22
	Setting Parameters for a Selected Document	24
	Setting Window Parameters	25
	Setting Toolbar Parameters for a Document Type	28
	Setting Action Parameters	28
	Action Definition	29
	Creating an Action	32
	Setting Popup Menu Parameters	32
	Popup Definition	33
	Creating a Popup Menu	34
	Adding a Popup Item	35
	Adding a New Popup Submenu	35
	Setting Dialog Parameters	35
	Dialog Definition	36
	Creating a Dialog Box	38
	Setting Data Parameters	39
	Data Definition	40
	Generating Parameters	40
	Parameters Command	40
	GUI Action Summary	44
Chapter 3	Implementing an Application	47
	How Application Framework Functions	47
	Option Files	49
	Main File	50
	Implementation of a Document Class	50
	New Document	51
	Serialization	51
	Commands	53

- Undo / Redo / Repeat Actions54
- Reflecting Changes Made In the Data to Associated Views55
- Implementation of a Document View Class.....56**
- Interactions.....57
- Chapter 4 Application Framework Interfaces 61**
- The Interface Mechanism61**
- Declaring an Interface62**
- Naming Convention for Macros62**
- Chapter 5 Actions..... 65**
- Activating an Action Event65**
- Processing an Action Event66**
- Index67**

About This Manual

This *User's Manual* describes the Application Framework package of IBM® ILOG® Views. Application Framework is a library designed to simplify the task of developing your graphical user interface (GUI) for applications based on the IBM ILOG Views Component Suite of C++ libraries for graphics creation and control.

What You Need to Know

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

Manual Organization

This manual contains the following chapters:

- ◆ Chapter 1, *Introducing IBM ILOG Views Application Framework* provides an overview of the document/view architecture and other features of the Application Framework package of IBM® ILOG® Views.

- ◆ Chapter 2, *Using the Application Framework Editor* describes the Application Framework Editor, itself an easy-to-use GUI.
- ◆ Chapter 3, *Implementing an Application* provides the steps and classes necessary to incorporate the document/view architecture of Application Framework.
- ◆ Chapter 4, *Application Framework Interfaces* describes how to incorporate the interface mechanism of Application Framework.
- ◆ Chapter 5, *Actions* describes how to activate and process actions under Application Framework.

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 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;
```

Introducing IBM ILOG Views Application Framework

The IBM® ILOG® Views Application Framework provides an easy-to-use graphics user interface (GUI) for defining the user interface for an application. This chapter provides an overview of the IBM ILOG Views Application Framework package. It includes the sections:

- ◆ *What is Application Framework*
- ◆ *The Document/View Architecture*

What is Application Framework

Application Framework is a library that lets you develop complete applications, such as the one shown in Figure 1.1:

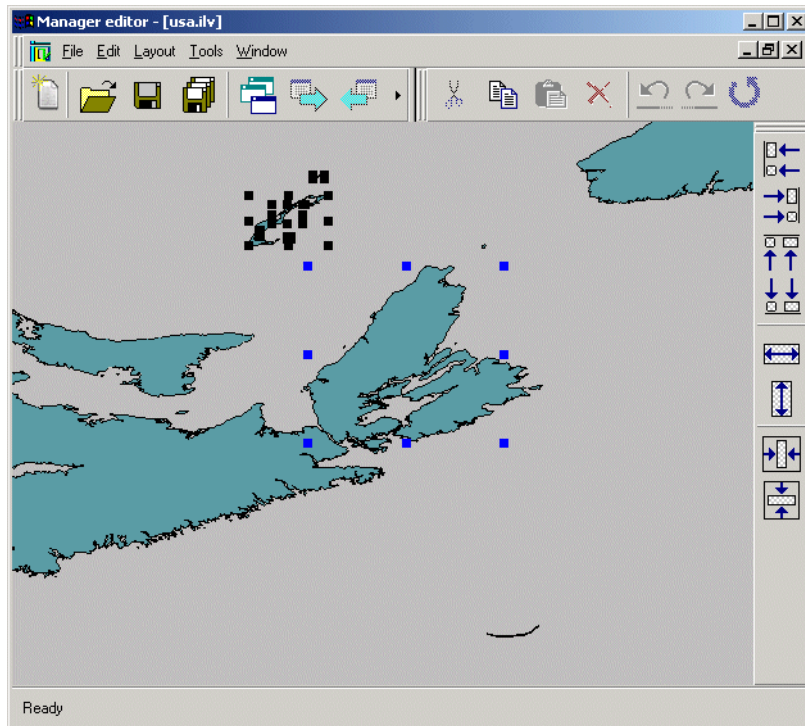


Figure 1.1 Application Developed with Application Framework

It provides a tool called the Application Framework Editor, which allows you to edit the application graphically: all menus, bars, actions, dynamic menus, and so on, are specified using this tool. Chapter 2 describes in detail how to start up and use the Application Framework Editor.

Application Framework also provides a mechanism that allows its objects to track and process GUI events. This mechanism will be looked at in Chapter 4, *Application Framework Interfaces*.

The Document/View Architecture

Application Framework is built on a Document/View architecture, common to most Windows applications. In this type of architecture, the application is a *frame window* holding toolbars and menus, that allows you to edit several documents at the same time. This frame window manipulates *documents* (data that is opened using menu items such as File > Open, File > New, and so on) that the user can edit inside *views*, which are usually created in a frame window.

For example, a document in Microsoft Excel is a table in memory loaded from an `.xls` file, and the views that can display and modify this document are sheets or charts.

Warning: *In Microsoft applications, the term document is used for both the data in memory and the view that lets the user edit the data.*

Chapter 3, *Implementing an Application* describes the document/view architecture in more detail.

Using the Application Framework Editor

The IBM® ILOG® Views Application Framework provides an easy-to-use graphics user interface (GUI) for defining the user interface for an application.

Starting Up the Application Framework Editor

On initial startup, the Application Framework Editor shows this dialog box:

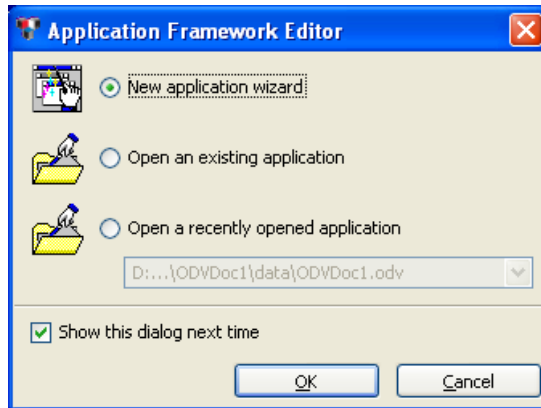


Figure 2.1 The Application Framework Editor Wizard

This dialog box has the following options:

- ◆ New application wizard - to begin creating a new application. See *Creating a New Application*.
- ◆ Open an existing application - to display the browse dialog box for selecting an existing application to open. See *Creating and Configuring an Options File (.odv file)*.
- ◆ Open a recently opened application - to quickly select an application from the drop-down list. See *Application Framework Editor Main Window*.

You can choose to bypass this screen by deselecting the “Show this dialog next time” option. In this case, you are taken directly to the Application Editor main window.

Application Framework Editor Main Window

The Application Framework Editor main window displays a menu bar, action toolbar, status bar, an Application Components palette, and a multidocument workspace. The startup window is shown in Figure 2.2.

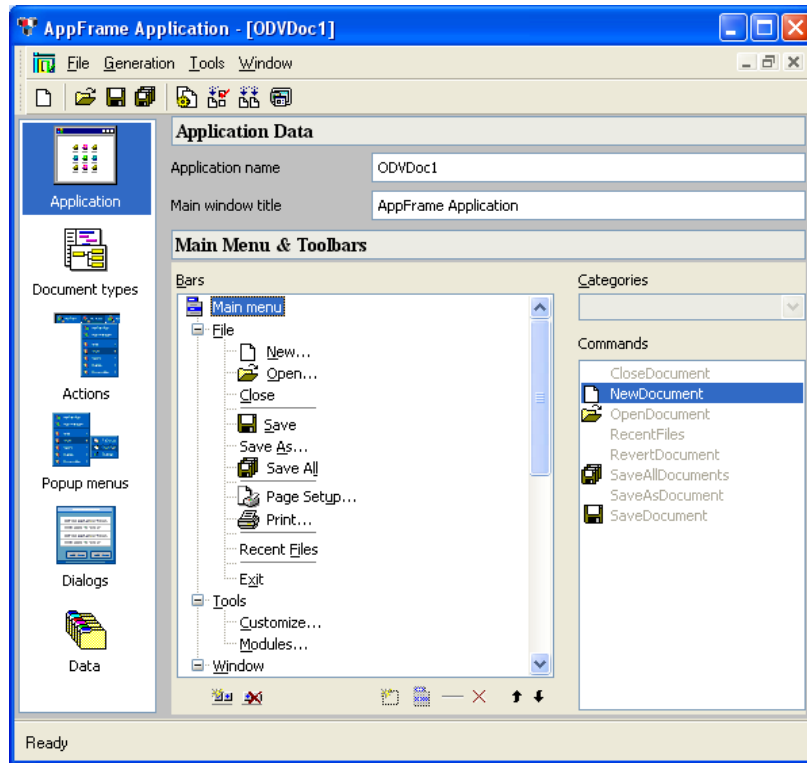


Figure 2.2 Application Editor Main Window

Components Palette

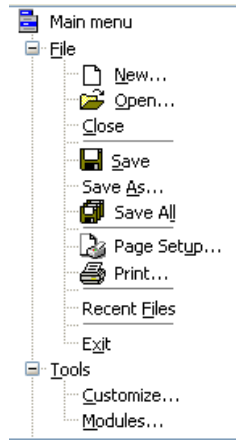
The Application Components palette on the left allows you to select the application entity you are editing:

- ◆ Application
- ◆ Document types
- ◆ Actions
- ◆ Popup menus
- ◆ Dialogs - for dialog boxes and windows
- ◆ Data

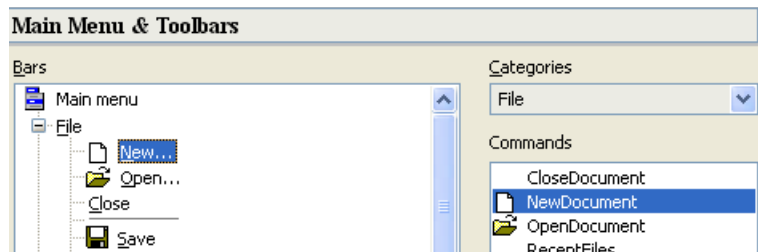
Workspace

The workspace on the right displays and allows you to set the parameters of the selected entry for each of the application entities.

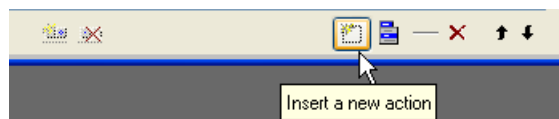
- ◆ A hierarchical tree in the middle area of the workspace allows you to select the item location to be edited or added to. For example:



- ◆ Selecting an item in the tree activates other possible parameter entry fields in the workspace. The fields are tailored to the specific operation. For example:











- ◆ The workspace toolbar at the bottom of the workspace allows you to easily select the operations tailored to the current workspace. For example:



The workspace toolbar changes appropriately, and items are grayed/activated, with the type of application entity. Possible toolbar icons are shown in Table 2.1

Table 2.1 *Workspace Toolbar*

Toolbar Icon	Description
	Insert a new action, popup, or dialog item below the currently selected tree or item; or a new category, accelerator, or other item to a list.
	Insert a new popup menu.
	Insert a new separator
	Remove the currently selected item.
	Move the selected item up in the tree.
	Move the selected item down in the tree.
	Insert a new toolbar.
	Remove the selected toolbar.

Creating a New Application

To develop an application with Application Framework, follow these basic steps:

1. Launch the Application Framework Editor and edit all the application options. Edit the different menus and toolbars that will appear in the application, describe all the document types that the application will be able to open, and so on. These items are saved in an options file that is read by the generated application, when initializing.
2. Generate the application code, using the Application Framework Editor.
3. Complete the generated code:
 - To manage the data (the document):
 - serialize the data,

- add accessors.
- To display and edit the data (the view):
 - initialize the view according to the data (an example of this is filling a tree gadget),
 - manage commands generated by the user events on views.

Note: You can modify the application options described in Step 1 at any time during the development of the application. You are not required to regenerate the code when you modify these options.

The remainder of this chapter describes the general navigation and operational features of the Application Framework Editor. For step-by-step instructions refer to the tutorial sample for Application Framework.

Selecting a Document Type

The first step in creating a new application is to select the document type. When you begin a New application, the Select a document type dialog box appears:

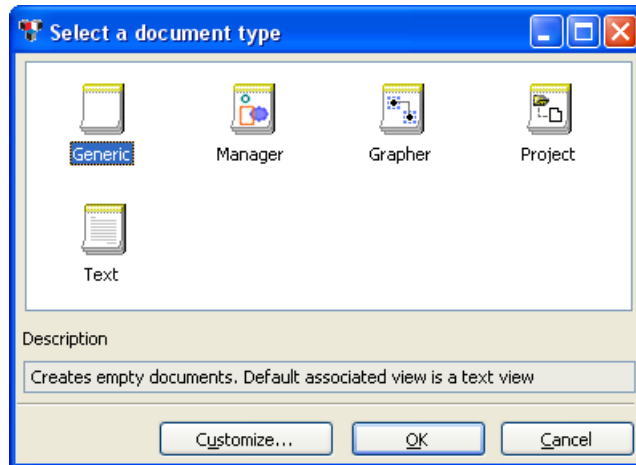







Figure 2.3 Select a Document Type

Several predefined types of documents are made available. Each type of document defines convenient methods for manipulating its data and is preassociated with a specific view.

The document types are described further in Table 2.2.

Table 2.2 Selection of Document Type

Document Type	Use To:	Description
	Create a generic application	The Generic document type does not make any assumptions about the type of document. This is the choice for most applications.
	Create a manager application	The Manager document type deals with <code>IlvGraphic</code> objects inserted in an <code>IlvManager</code> object.
	Create a grapher application	The Grapher document type deals with <code>IlvGraphic</code> objects inserted in an <code>IlvGrapher</code> as nodes or links.
	Create a project application	The Project document type is an organization of files in folders and subfolders.
	Create a text application	The Text document type is any text document.

After you select a document type, the Application Framework Editor main window appears. See *Application Framework Editor Main Window*.

Creating and Configuring an Options File (.odv file)

Application Framework stores all parameters that describe an application in an options file (.odv extension).

The Application Framework Editor opens a new .odv file whenever you create a new application (New from the File menu, toolbar, or initial wizard) and select a document type.

Setting Application Parameters

The Application Framework Editor is used to set your application parameters when *Application* is selected from the Application Framework Editor Palette.

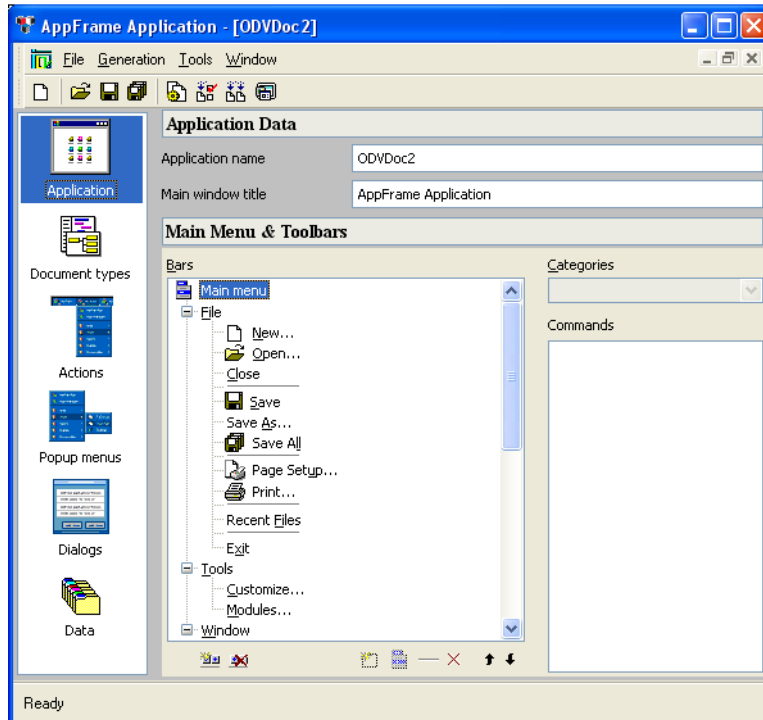



Figure 2.4 Application Selected from Palette

In the Application Data workspace of the Application Framework Editor you set:

- ◆ The application name in the *Application name* field. By default, this name is used to create the directory and the project name. The default name shown in Figure 2.2 is ODVDoc1.
- ◆ The main window title in the *Main window title* field. This is the name that will appear as the title in your application.


Adding Menu Items

You add menu items in the Main Menu & Toolbars section of the workspace when *Application* is selected from the Application Framework Editor Palette.

1. Select an item in the "Main menu" tree where you want to insert a new item. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Categories" combo box and "Commands" list.


If you want to modify a menu item, select the item and modify it by selecting the new action in the "Commands" list.

If you want to insert new commands, refer to *Creating an Action*.

To remove an item in the main menu bar, select the item to remove and click the delete button .


Adding Toolbar Items

You add toolbar items in the Main Menu & Toolbars section of the workspace when *Application* is selected from the Application Framework Editor Palette.

1. Select an item in the "Standard" tree where you want to insert a new button in the toolbar. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Categories" combo box and "Commands" list.

If you want to modify a menu item, select the item and modify it by selecting the new action in the "Commands" list

If you want to insert new commands, refer to *Creating an Action*.

To remove an item from the toolbar, select the item to remove and click the delete button .

Setting Document Parameters

The Application Framework Editor is used to set your document parameters when *Document types* is selected from the Application Framework Editor Palette.

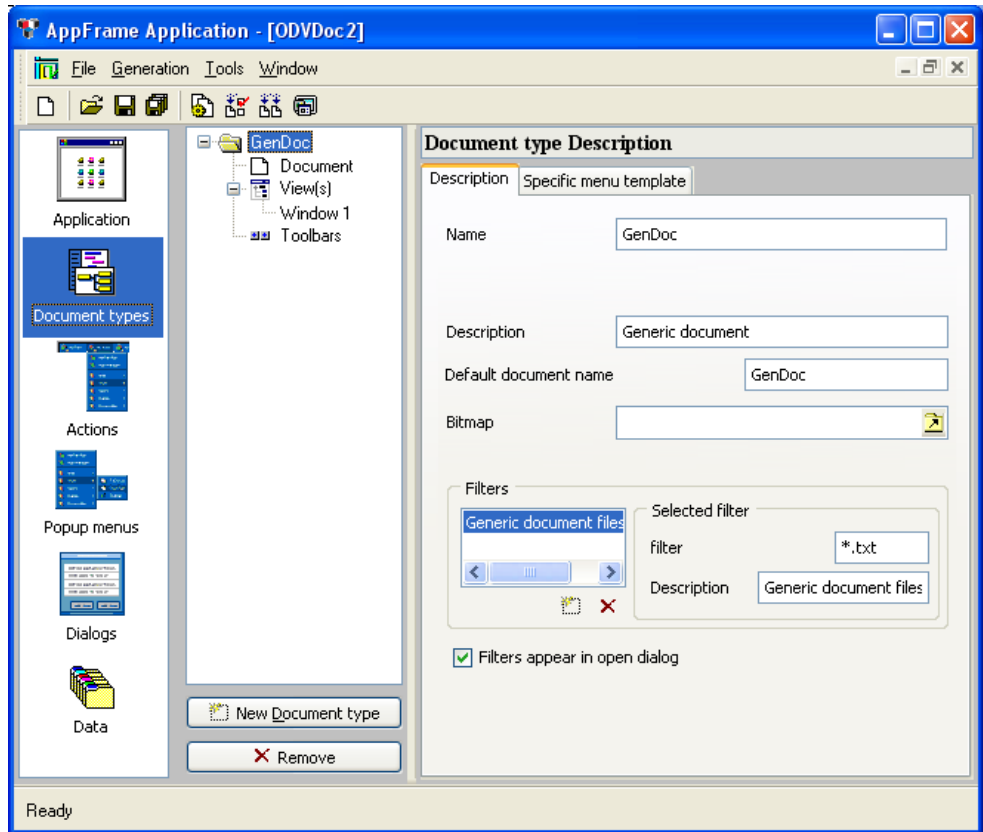



Figure 2.5 'Document Types' Selected from Palette

The middle column shows the document tree, headed by the document type (GenDoc in Figure 2.5; or Grapher, Project, Text, or Manager depending on the chosen type). This column shows the document types that the application can handle. You can add many document types with the "New Document Type" button .

The right column allows you to change parameters of the selected items in the middle columns as described below.

Setting General Document Parameters

When GenDoc is selected in the tree, the workspace has the following tabs:

- ◆ *Description* tab contains basic information about the document type.

The screenshot shows a dialog box titled "Document type Description". It has a tab labeled "Description". The dialog contains the following fields and controls:

- Name:** A text box containing "GenDoc".
- Description:** A text box containing "Generic document".
- Default document name:** A text box containing "GenDoc".
- Bitmap:** An empty text box with a browse button (represented by a folder icon).
- Filters section:**
 - A list box containing "Generic document files".
 - A "Selected filter" field containing "*.*".
 - A "Description" field containing "Generic document files".
 - Buttons for navigating the list (back, forward, and a central button).
 - Icons for adding and removing filters.
- Filters appear in open dialog:** A checked checkbox.

Figure 2.6 'Description' Tab (Document Type Description)

- **Name:** This name is used when you want to retrieve the document template that can create this type of document.
- **Description:** The description of the document type.
- **Default Document Name:** This name is used when a document of this type is created. When created, the document has this name.
- **Bitmap:** This bitmap associated to the document type.
- **Filters section:** This section allows you to define the elements that will appear in the open document dialog box of your application. These elements will be used in the "Files of types" section in the open document dialog box .
- **Filter:** The extension of the document files. The form of this field should be * .xxx where xxx is the extension.
- **Description:** Description of the filter.
- **Filters appears in open dialog:** If the check box is checked, these filters will appear in the open dialog box of your final application.

- ◆ *Specific menu template* tab sets the menu visibility feature for a document.

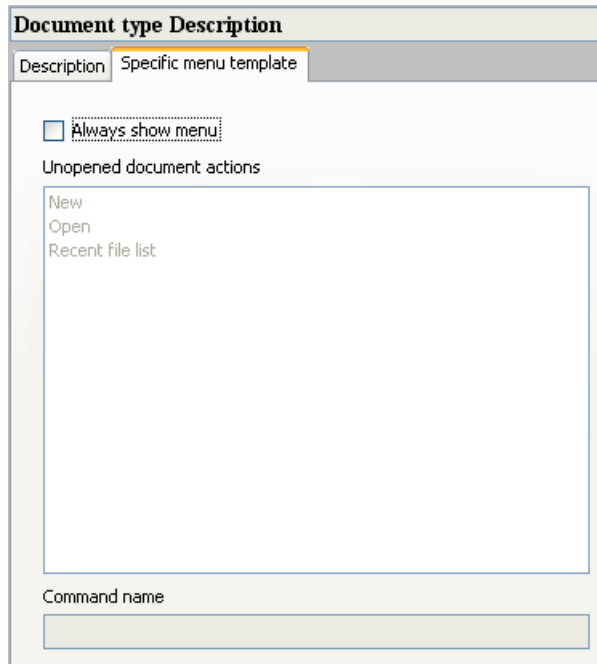


Figure 2.7 'Specific Menu Template' Tab

- ◆ Always show menu: When the toggle is checked, the specific menu and toolbar are added even if a document of this type is not active.

Setting Parameters for a Selected Document

When the `Document` item for a document type is selected in the tree, the workspace has the following fields:

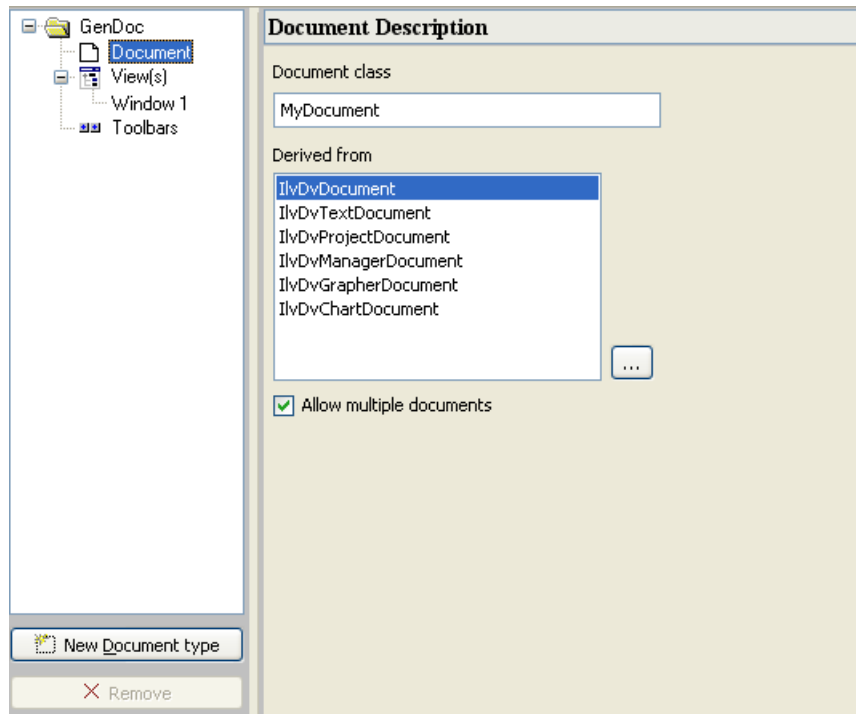


Figure 2.8 'Document Description'

- ◆ The Document class specifies the name of the class used during the code generation.
- ◆ You can specify the parent class by choosing one item in the "Derived from" string list which is filled by a predefined class.
- ◆ The "Allow multiple documents" check box specifies if your application can handle many document of this type or only one.

Setting Window Parameters

When the Window item for a document type is selected in the tree, the workspace has the following tabs:

- ◆ View tab allows you to define the class of the views on the document.

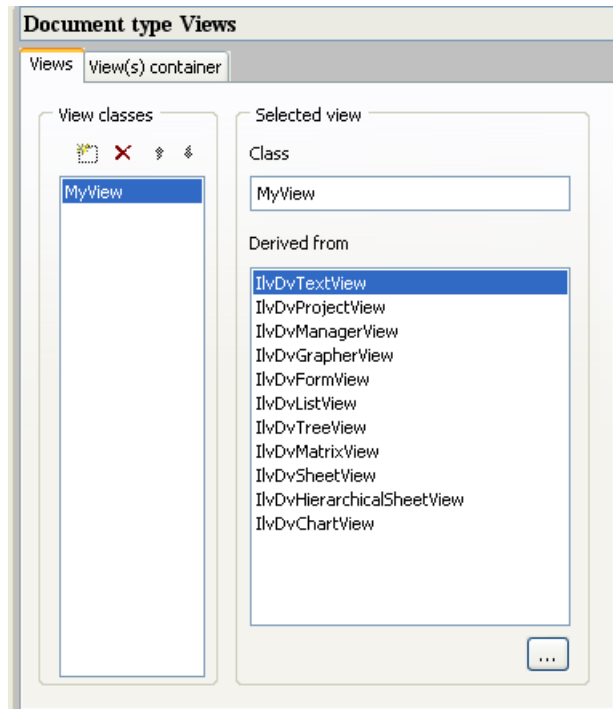


Figure 2.9 'Views' Tab

- In the Class text field, you specify the class name that will be used during the code generation.
- In the "Derived from" list, you select the class from which the selected view class is to be derived.

- ◆ *View(s) container* tab provides additional parameters for the container that will contain the view.

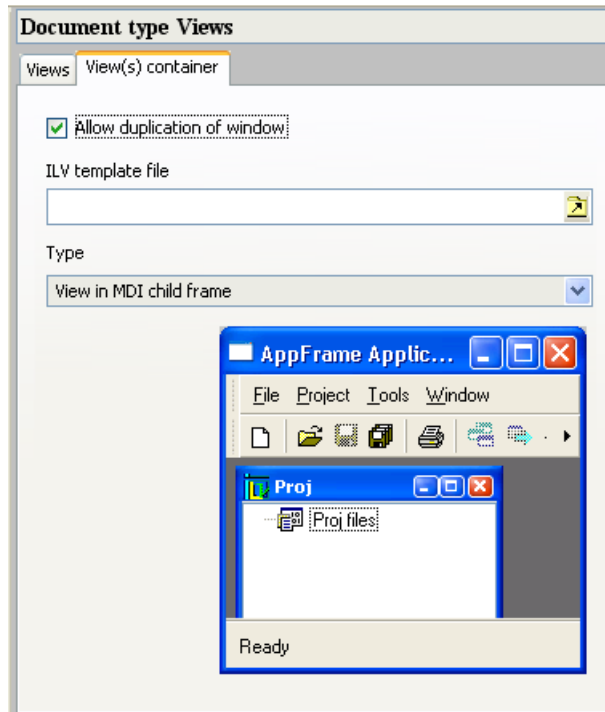


Figure 2.10 'View(s) Container' Tab

- "Allow duplication of window" specifies if your application can handle only one or many views on the same document.
- In the "Type" combo box, you can choose the initialize configuration of your window. You can choose between:
 - View in MDI child frame
 - View in MDI maximized child frame
 - Docked at left
 - Docked at right
 - Docked at top
 - Docked at bottom
 - Docked in a float window

- For all docked configurations, you can specify a method in the "Show/hide action" text field which will be called when this window appears or disappears. This text field is displayed only for docked configurations.

Setting Toolbar Parameters for a Document Type

When the `Toolbars` item is selected in the tree, the workspace allows you to define or change a specific toolbar that is to be displayed only when a document of this type is active. For editing this toolbar see *Adding Toolbar Items*.

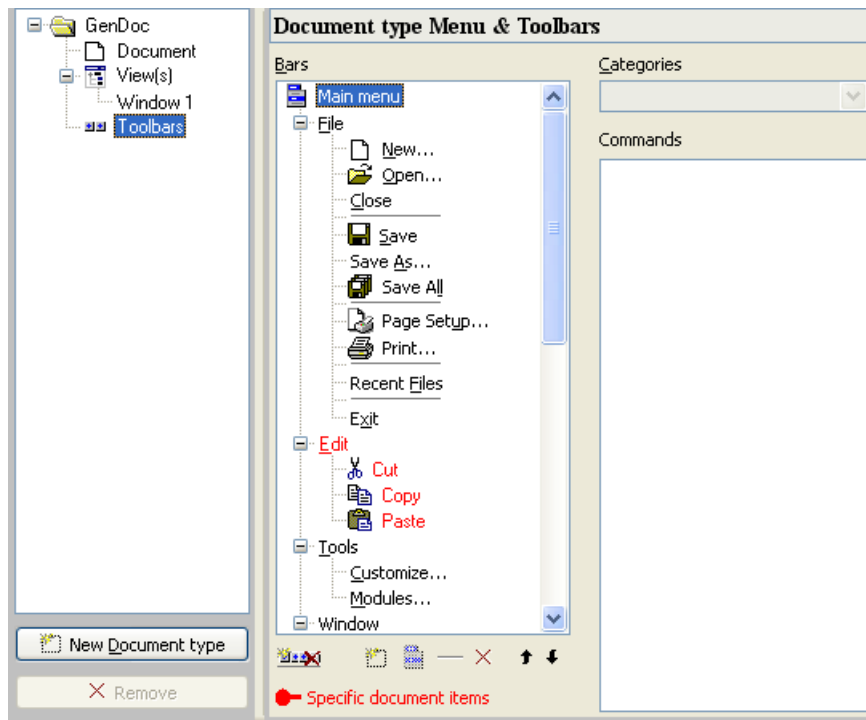


Figure 2.11 'Document Type Menu & Toolbars'

Setting Action Parameters

The Application Framework Editor is used to set your action parameters when *Actions* is selected from the Application Framework Editor Palette.

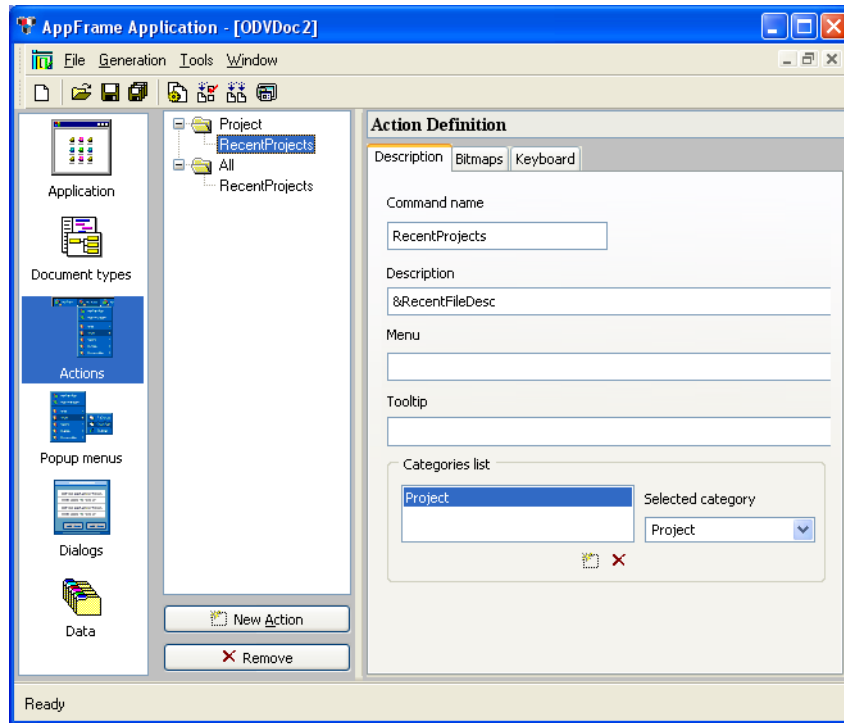


Figure 2.12 'Actions' Selected from Palette

The middle column shows the Actions tree. By clicking either of the `RecentProjects` in the tree, you display the Action Definition.

Action Definition

The Action Definition workspace has the following tabs:

- ◆ *Description* tab contains basic information about the action.

Figure 2.13 ‘Description’ Tab of Action Definition

- **Command Name:** This name is created by the user to identify the action.
- **Description:** The user can enter a description of the action.
- **Menu:** The name of the item appearing in the menu (for example New, Open, Save, and so forth).
- **Tooltip:** The name of the item appearing in the tooltip text (for example “New (Ctrl+N)”).
- **Categories list:** shows the category of the action.
- **Selected Category:** lists the current categories that can be selected (Application, File, Project, and so forth).

The Description, Menu, and Tooltip items can be text, or they can be a message identifier of the form *&identifier* for text contained in a `.dbm` file. (For information on the `.dbm` file type see the *IBM ILOG Views Foundation User’s Manual*.)

- ◆ *Bitmaps* tab shows characteristics of the bitmap icon associated with the action. This icon appears in the menu, toolbar, and tree lists with the action name.

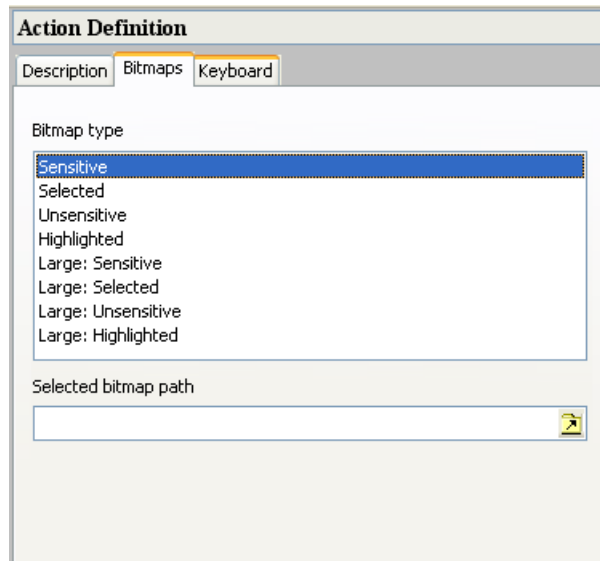


Figure 2.14 'Bitmaps' Tab of Action Definition

- **Bitmap Type:** You can define a set of bitmaps for each different type in the string list. These bitmaps will be used depending on the status of the action.
- **Selected bitmap path:** The path where the icon is found for the selected bitmap type.

- ◆ *Keyboard* tab shows the keyboard shortcut for the action.

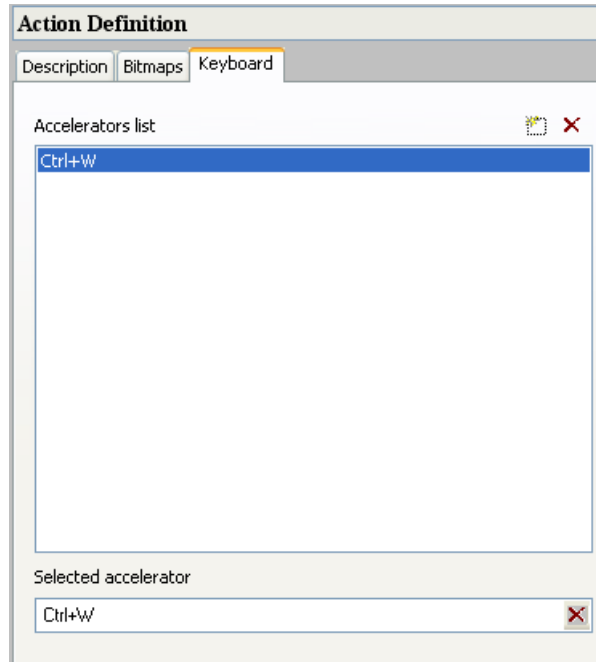



Figure 2.15 'Keyboard' Tab of Action Definition

- Accelerators list: To add an accelerator click the Add button  and use the 'Selected accelerator' field, for example 'Ctrl+W' is added as a default first accelerator.
- Selected Accelerator: The current accelerator or keyboard shortcut. To change the shortcut, click in the field, and then type the sequence of the sortcut on your keyboard.

Creating an Action

For complete details on implementing actions, see Chapter 5, *Actions*.

Setting Popup Menu Parameters

The Application Framework Editor is used to set your popup menu parameters when *Popup menus* is selected from the Application Framework Editor Palette.

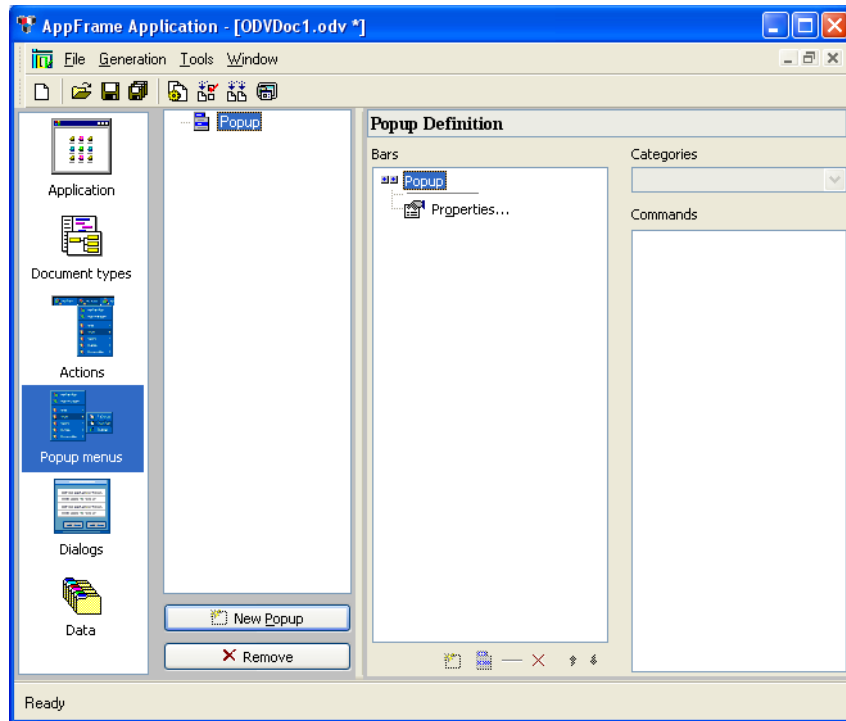


Figure 2.16 'Popup Menus' Selected from Palette

The Popup Definition workspace becomes active when you begin adding a popup menu.

Popup Definition

The Popup Definition workspace allows you to define popups that will be accessible from your application.

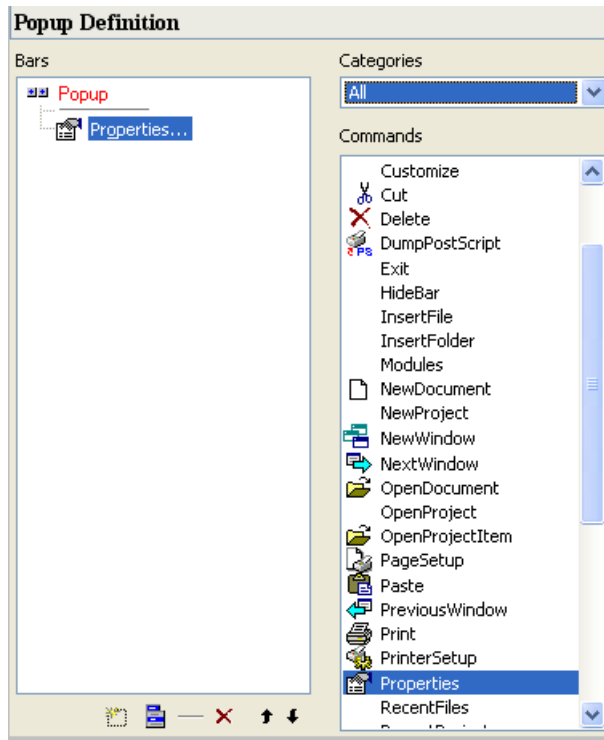



Figure 2.17 Popup Definition

- ◆ The tree shows the layout of the popup menu. When an item of the popup is selected, the remaining fields become activated.
- ◆ Categories: A list showing the possible categories. You can select a category of action to retrieve an action more easily. When the All category is selected (as shown), all commands are displayed alphabetically in the Commands list.
- ◆ Commands: The possible commands in the selected category.

Creating a Popup Menu


To begin defining a new popup menu, click the New Popup button  below the main tree. This column shows a new item in the tree which is the new popup menu created (see Figure 2.16). The created popup has a default layout with two items (separator and Properties items), but you can change this layout.

The default name of the popup menu is `Popupxx` where `xx` is an incremental number when you insert more than one popup menu. You can change the name of this popup by selecting the root item in the Popup Definition window, pressing the F2 key, and then typing the new name.

In your code, you can retrieve this popup by using the `IlvDvApplication::readPopup(const IlvSymbol*)` function.


In the Popup Definition window, you can define the layout of the popup by adding or removing items. For adding items to the popup menu see *Adding a Popup Item*.

Adding a Popup Item

1. Select an item in the popup layout where you want to insert a new item. The item will be inserted after this item.
2. Click the "Insert a new action" button .
3. Modify the inserted item by choosing the associated action in the "Commands" list.


If you want to modify a popup item, select the item and modify it by selecting the new action in the "Commands" list.

If you want to insert new commands, refer to *Creating an Action*.

To remove an item in the popup, select the item to remove and click the delete button .

Adding a New Popup Submenu

The Popup Definition workspace allows you to submenus in the popup.

- ◆ In the Popup Definition tree, select the item where you want to insert the submenu. The submenu will be inserted after this item.
- ◆ Click the "New" popup menu button  in the Popup Definition toolbar.
- ◆ Modify the label of the popup item (use the F2 accelerator) and then enter the new label.
- ◆ Modify the submenu by adding or editing the items of the submenu (see *Adding a Popup Item*).

Setting Dialog Parameters

The Application Framework Editor is used to set your dialog box parameters when *Dialogs* is selected from the Application Framework Editor Palette.

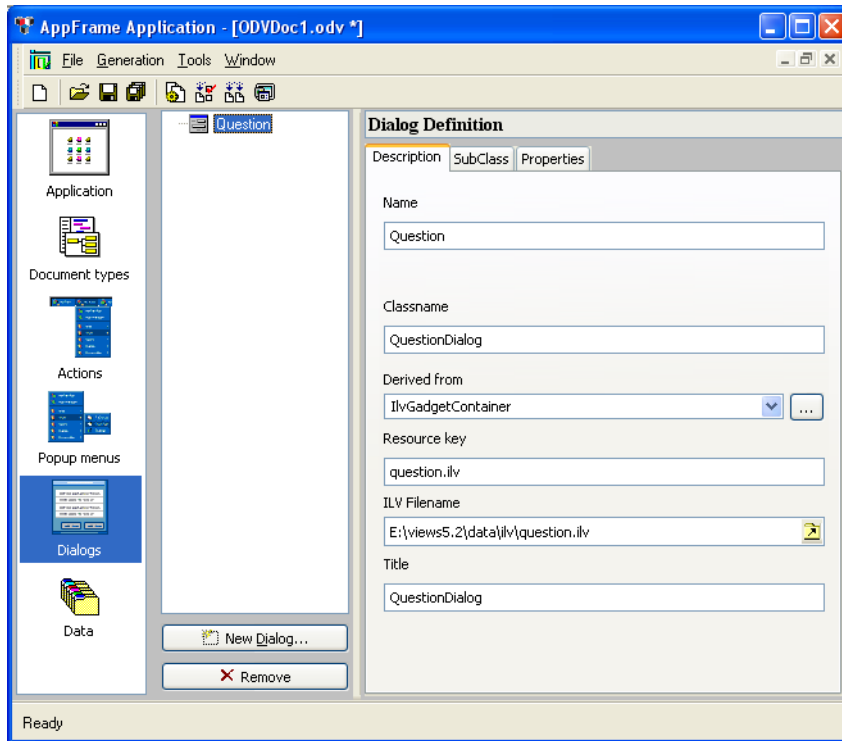


Figure 2.18 'Dialogs' Selected from Palette

The Dialog Definition workspace becomes active when you begin adding a dialog.

Dialog Definition

The Dialog Definition workspace allows you to define dialog box properties. The dialog box must first be defined in IBM ILOG Views Studio.

The Dialog Definition workspace has the following tabs:

- ◆ *Description* tab contains basic information about the dialog box.

The screenshot shows a 'Dialog Definition' window with three tabs: 'Description', 'SubClass', and 'Properties'. The 'Description' tab is active. It contains the following fields:

- Name:** A text box containing 'Question'.
- Classname:** A text box containing 'QuestionDialog'.
- Derived from:** A dropdown menu showing 'IlvGadgetContainer' with a blue arrow and a '...' button to its right.
- Resource key:** A text box containing 'question.ilv'.
- ILV Filename:** A text box containing 'E:\views5.2\data\ilv\question.ilv' with a folder icon button to its right.
- Title:** A text box containing 'QuestionDialog'.

Figure 2.19 'Description' Tab of Dialog Definition

- **Name:** This name is by default the name of the `.ilv` file, for example `Question` when the file loaded is `question.ilv`.
- **Classname:** This name is by default the Name and the word `Dialog`, for example `QuestionDialog`. This name will be used during the code generation.
- **Derived From:** The IBM ILOG Views class from which the dialog is derived. Select from the list.
- **Resource Key:** The name of the resource used to reread this file, by default the `.ilv` file name.
- **ILV Filename:** The full path name of the `.ilv` file that was loaded.
- **Title:** The title that appears in the Windows title bar. By default it is the Classname.

- ◆ *Subclass* tab contains the name of the dialog's subclass. This entry is optional.

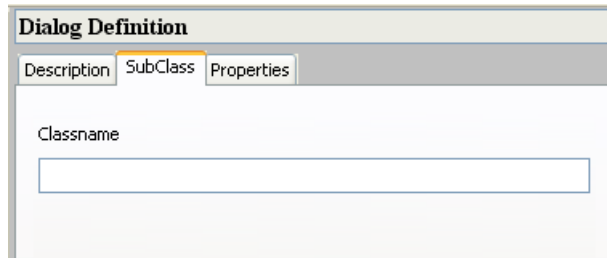


Figure 2.20 'SubClass' Tab of Dialog Definition

- ◆ *Properties* tab allows you to specify standard properties of the dialog box. Choose any or all of these properties.

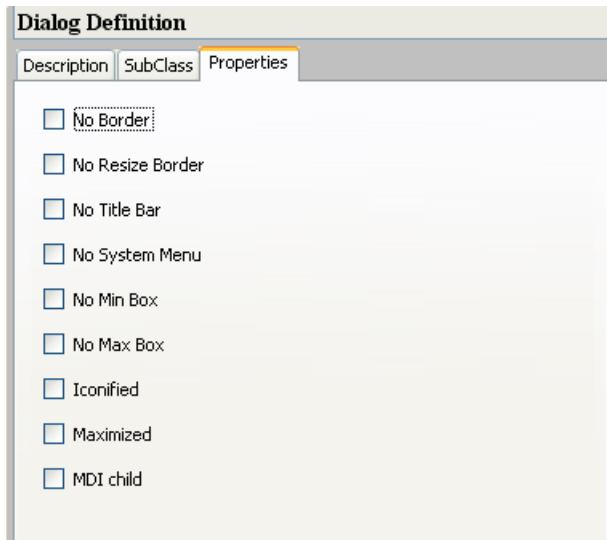



Figure 2.21 'Properties' Tab of Dialog Definition

Creating a Dialog Box

To begin a definition, click the New Dialog button . You are requested to open the predefined `.ilv` file.

Note: To create a dialog box in the Application Framework Editor, you must define a dialog box in IBM ILOG Views Studio and save it (`.ilv` file). This name is requested when creating a dialog box in the Application Framework Editor.

The middle column shows the Dialog tree, with the Dialog Definition on the right. Complete the information for the Dialog Definition tabs (for details see *Dialog Definition*).

Setting Data Parameters

The Application Framework Editor is used to set your data parameters when *Data* is selected from the Application Framework Editor Palette.

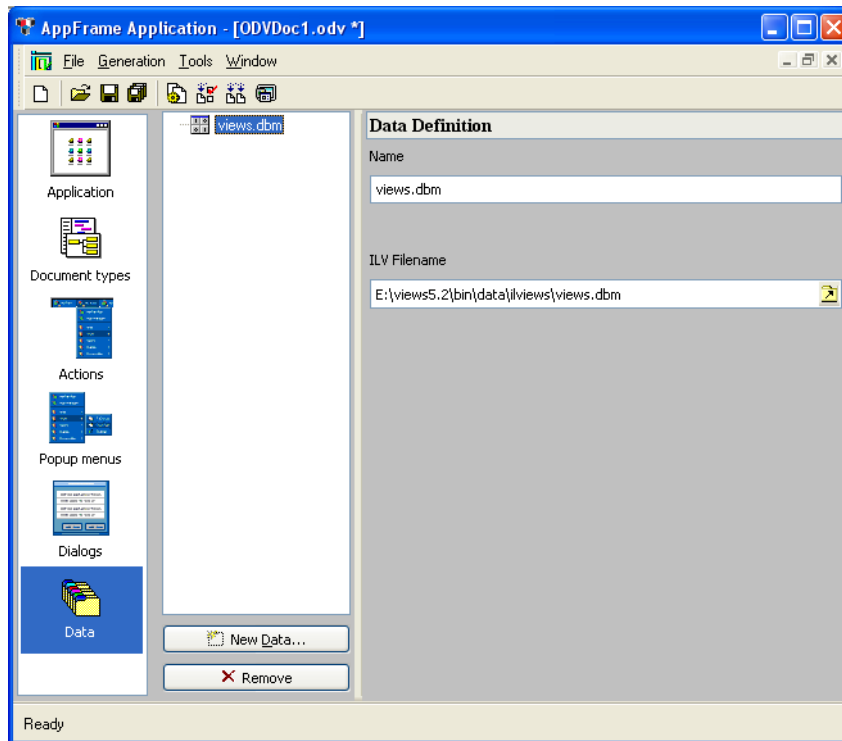




Figure 2.22 'Data' Selected from Palette

You can use this feature to add data files to the application executable. They can be any data files: .dbm, bitmaps, .ilv, or user data files that are not otherwise included.

The Data Definition workspace becomes active when you begin adding a data file by clicking the New Data button .

Data Definition

The Data Definition workspace (see Figure 2.22) allows you to define the data properties needed to include the file.

- ◆ **Name:** The name to be used to reread the file for retrieving the data. The default is the name of the file that was loaded. You can change this name by editing the text field.
- ◆ **ILV Filename:** The full path name of the file. This path name can be changed by clicking  and selecting a new path.

Generating Parameters

After you have defined the application parameters in the Application Framework Editor, you must generate it.

The Generation menu provides commands to generate the application.

Parameters Command

For initial generation, when you select Generation -> Parameters, it displays the Project Generation window.

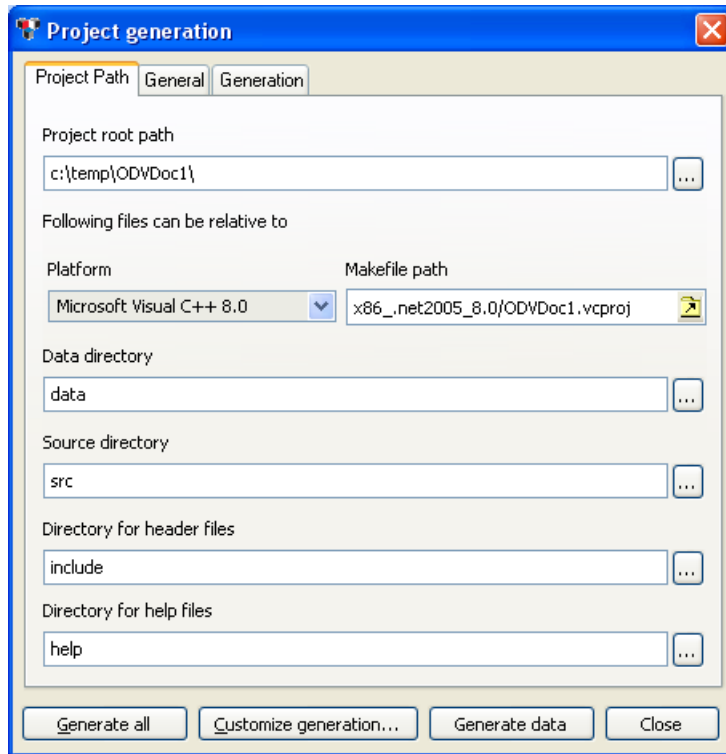


Figure 2.23 Project Generation window, Project Path tab

This window has three tabs:

- ◆ *Project Path* tab (see Figure 2.23) contains the fields:
 - Project Root Path: The root path where the project is saved. The following paths can be relative to this root path.
 - Platform: The platform for the makefile.
 - Makefile Path: The path for the makefile, based on the Platform selection.
 - Directories for: Data, Source, Header files, and Help files. These are all given defaults but can be changed.

- ◆ *General* tab allows you to set general information fields.

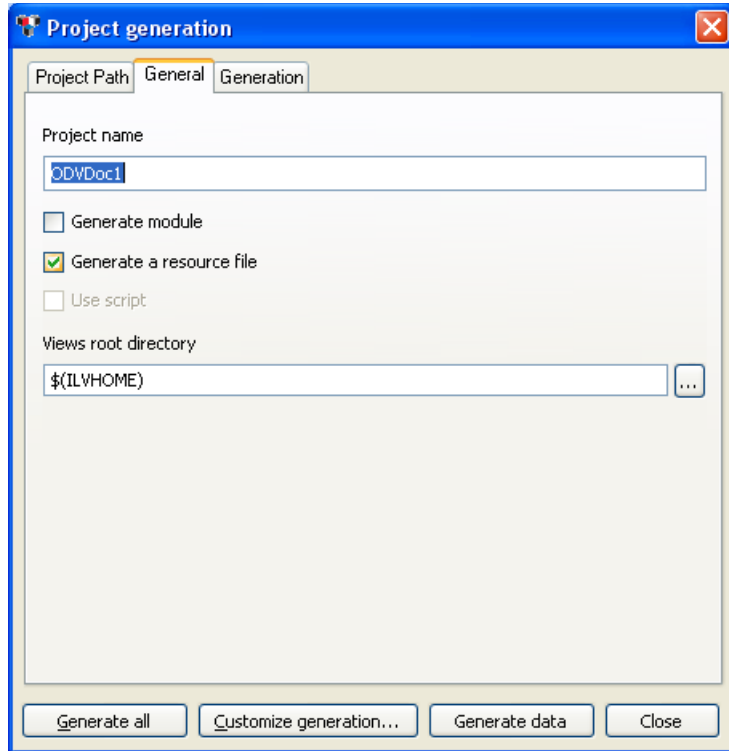


Figure 2.24 'General' tab (Project Generation)

- ◆ *Generation* tab provides information about the project generation.

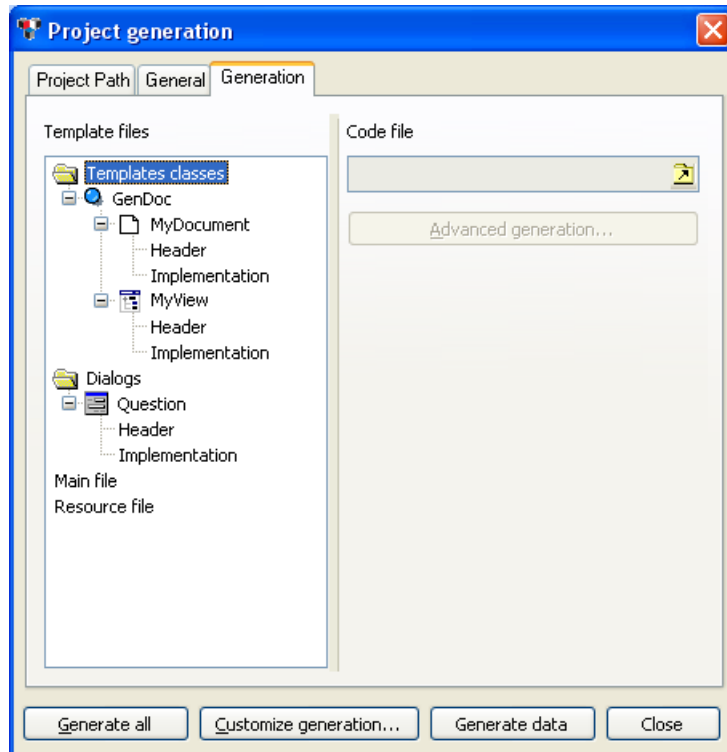


Figure 2.25 'Generation' tab (Project Generation)

Generate All

Use Generate All to generate all files of your application.

Important: This operation replaces all existing generated files of your application. A dialog asks you for confirmation before proceeding.

Custom Generation

Use Custom Generation to generate just one or selected portions of the application. This can be done when adding a new dialog box, for example.

Generate Data

Use Generate Data for updates that do not require changing the source code or makefiles after the initial generation of the application. For example, it can be used to add an action, a document type, a popup, or a new data file.

GUI Action Summary

Table 2.3 Menu and Toolbar Operations











Action	Toolbar Icon	Menu Operation	Comments
File Operations			
Create a new project		File>New	
Load an .odv file		File>Open	
Close the current .odv file		File>Close	
Save an .odv file		File>Save File>Save As	For Save As, type a new name including file extension.
Save all open .odv files		File>Save All	
Generation Operations			
Set project generation parameters		Generation>Parameterst	
Generate the entire application.		Generation>Generate all the application	
Generate specific files in the current application.		Generation>Generate specific files...	Opens the Custom Generation dialog box.
Generate data.		Generation>Generate data	Displays the generation report log.
Tools			
Customize application		Tools>Customize	Opens the Customize window.
Insert and remove modules		Tools>Modules	Opens the Insert/Remove modules dialog box.
Script a project		Tools>Script project	Creates or opens a script project file (.spj).
Window Operations			

Table 2.3 *Menu and Toolbar Operations (Continued)*

Action	Toolbar Icon	Menu Operation	Comments
Start a new window		Window>New window	
Display the next window		Window>Next Window	
Display the previous window		Window>Previous Window	
Cascade all document views		Window>Cascade Windows	
Tile all document views horizontally		Window>Tile Horizontally	
Tile all document views vertically		Window>Tile Vertically	
Quit the Application Framework Editor		File>Exit	Asks about unsaved files before exiting.

Implementing an Application

This chapter discusses how to implement an application under Application Framework, including a description of the classes and files required. It is divided as follows:

- ◆ *How Application Framework Functions*
- ◆ *Option Files*
- ◆ *Main File*
- ◆ *Implementation of a Document Class*
- ◆ *Commands*
- ◆ *Implementation of a Document View Class*

How Application Framework Functions

Application Framework is built on a Document/View architecture (see *The Document/View Architecture*). Figure 3.1 illustrates the different classes that the Document/View mechanism relies upon.

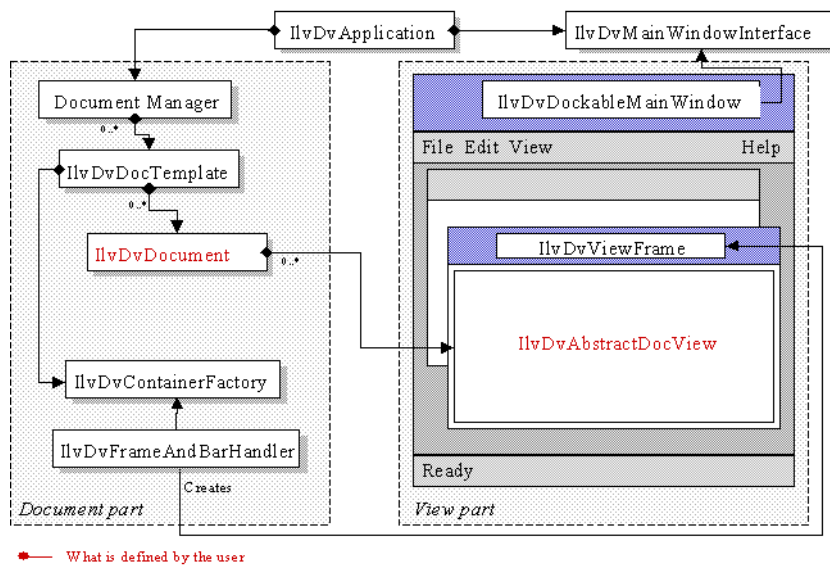


Figure 3.1 Document/View Classes

All the classes shown in the figure, except the `IlvDvApplication` class, the `IlvDvDocument` hierarchy, and the `IlvDvDocViewInterface` hierarchy, are hidden from the developer. Instances of these classes are automatically created according to the application options that are read while the application is initializing.

The code of an Application Framework application consists of:

- ◆ **Option Files:** At least one option file, which is edited using the Application Framework Editor.
- ◆ **Main File:** A main file containing the main entry point of the program, which must instantiate an `IlvDvApplication` (or a derived class) object. This file is generated by the Application Framework Editor and must only be completed in very specific cases, as shown in the `Text` sample.
- ◆ **Implementation of a Document Class:** Files implementing a document class, which is a subclass of `IlvDvDocument`.
- ◆ **Implementation of a Document View Class:** Files implementing a document view class, which is a subclass of `IlvDvDocViewInterface` class.

Option Files

While initializing, an Application Framework application reads three option files that contain data. This data can be the contents of menus and toolbars, the recently used file list, document templates, and so on.

The option files are the following:

- ◆ The Application Framework option file. Its path is `<ILVHOME>/data/ilviews/appframe/docview.odv` and it is contained by the `<ILVHOME>/data/res/appframe.rc` file.

This file contains the descriptions of the default actions (OpenDocument, SaveDocument, Cut, Copy, and so on), the default menus, and the description of the default toolbars.

- ◆ The application option file. The file path is given to the `IlvDvApplication` object using the `IlvDvApplication::setAppOptionsFilename` method.

This file is edited using the Application Framework Editor and contains the following information:

- Application name and title of the main window.
- Description of different document templates.
- The main menu and the toolbars, if different from the default ones stored in the Application Framework option file.
- Description of actions.
- User application data.
- ◆ User profile options file. Its default path is given as follows:

- Windows:

```
<Windows directory>/Profiles/<Username>/Application Data/  
<Application name>.odv
```

- UNIX:

```
$(HOME)/<Application name>.odv
```

To specify a different path, use the method:

```
IlvDvApplication::setUserOptionsFilename
```

This file contains the application data modified by the user the last time the application was run. It is mainly composed of:

- Most Recently Used file list.
- Position and size of the application main window.

- Positions and state (hidden or not) of all dockable toolbars.
- Customizing of toolbar contents.
- Customizing of actions.

Main File

When you create an Application Framework application, you must first create an `IlvDvApplication` object in the main procedure, the same way you create an `IlvApplication` object in a simple IBM® ILOG® Views application:

Note: *The main file is automatically generated using the Application Framework Editor.*

```
int
main(int argc, char* argv[])
{
    IlvDvApplication* app = new IlvDvApplication("", 0, argc, argv);
    IlvDisplay* display = app->getDisplay();
    if (!display || display->isBad()) {
        IlvFatalError("Couldn't create display");
        delete display;
        return -1;
    }
    // Adding the options file
    app ->setAppOptionsFilename((const char*)"myapp.odv");

    // Adding the data base file
    display->getDatabase()->read((const char*)"myapp.dbm", display);

    // Continue...
    application->run();
    return 0;
}
```

`IlvDvApplication` is a subclass of `IlvApplication` and features management of options data and the handling of menu and toolbar items, as well as actions and their states.

Most of all, this `IlvDvApplication` object is aware of all objects involved in the Document/View mechanism (see Figure 3.1). Similarly, all these objects are aware of the application object. The application object is useful, for example, when changing the state of an action from a document or from a document view.

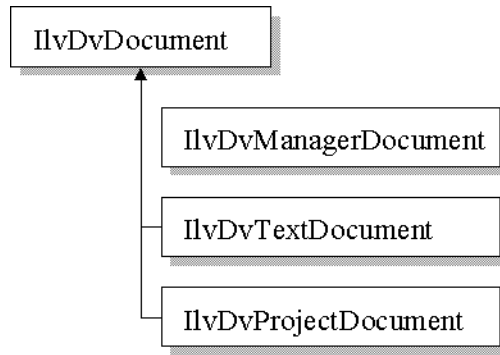
Implementation of a Document Class

A document class is derived from the `IlvDvDocument` class.

The document is *the user data*. The document class loads and saves data and also provides accessors that are used by document views to modify data.

Note: *The user data is similar to the Model View Controller (MVC) approach.*

Application Framework provides document classes that manage specific data, such as IBM ILOG Views managers, text buffers, or projects, as shown in the following inheritance tree:



New Document

A derived document class must override the `IlvDvDocument::initializeDocument` method.

It is called when the File > New command is executed to create a document.

The method must first call `IlvDvDocument::initializeDocument`. Then, it must initialize specific data.

Serialization

The `IlvDvDocument::serialize` method:

```
void IlvDvDocument::serialize(IlvDvStream& stream);
```

is called when a file is opened to create the document, if a call to `stream.isSaving()` returns false. Otherwise, the document must be saved.

Typically, the body of the method has the following form:

```
IlvDvDocument::serialize(stream);
if (stream.isSaving()) {
    // Here, write your persistent data
}
else {
```

```

    // Here, read data from stream
}

```

There are two ways of loading and saving data when using the parameter called stream.

One way is to use `istream` or `ostream` objects. These objects are given by a call to the `istream* getInStream() const` and `ostream* getOutStream() const` methods directly.

The other way, which is usually easier, is to use specific serialization methods, provided by the `IlvDvStream` class. Here are these methods:

◆ Operators

```

// Storing operators
IlvDvStream& operator<<(IlvInt i);
IlvDvStream& operator<<(IlvUShort w);
IlvDvStream& operator<<(IlvShort ch);
IlvDvStream& operator<<(IlvUInt u);
IlvDvStream& operator<<(IlvBoolean b);
IlvDvStream& operator<<(IlvFloat f);
IlvDvStream& operator<<(IlvDouble d);
IlvDvStream& operator<<(const IlvString& s); // 's' must not contain blanks

// Reading operators
IlvDvStream& operator>>(IlvInt& i);
IlvDvStream& operator>>(IlvUShort& w);
IlvDvStream& operator>>(IlvShort& ch);
IlvDvStream& operator>>(IlvUInt& u);
IlvDvStream& operator>>(IlvBoolean& b);
IlvDvStream& operator>>(IlvFloat& f);
IlvDvStream& operator>>(IlvDouble& d);
IlvDvStream& operator>>(IlvString& s);

```

◆ `void serialize(IlvString&, IlvBoolean betweenQuotes = IlvTrue);`

This method is a safe way of loading and saving strings. If the `betweenQuotes` parameter is set to true, the string is saved between quotation marks. This way it can contain blank spaces.

◆ `void serializeBitmap(IlvBitmap*&, IlvBoolean lock = IlvTrue);`

Serializes a bitmap path.

◆ Serialization of objects

When implementing user classes, it is recommended to derive from the `IlvDvSerializable` class. This class is an abstract interface that provides both a mechanism for safe downcasting and a serialization method:

```
virtual void serialize(IlvDvStream& stream);
```

- `void serializeObjects(IlvArray&);`

Load and save an array of `IlvDvSerializable` objects:

- `void writeObject(const IlvDvSerializable*);`

- `IlvDvSerializable* readObject();`
- ◆ `virtual void clean();`

This method is called to clean up the document data. It is only used for documents whose corresponding document type does not allow opening more than one document (SDI document types, typically project documents). When a user tries to create a new document while a document of the same type is already open, Application Framework does not delete the currently opened document to create another one. Instead, it cleans the open document (by calling this method) and reinitializes the document.

Commands

To modify the data of a document, it is recommended to use the Application Framework command mechanism, which provides the following advantages:

- ◆ Undo/Redo mechanism - The Undo, Redo, and Repeat actions are automatically processed, as well as their state.
- ◆ The modification state of a document is automatically managed. Adding a command to an unmodified document will mark the document as modified (a star will appear in the title of the frames that contain views associated with this document). Similarly, undoing this command will restore the unmodified state of the document (and will remove the star from the title of the same frames).
- ◆ Keeps a log of all modifications made to the document.

Consider the following document class:

```
class MyDocument
: public IlvDvDocument
{
...
    void setX(int x) { _x = x; }
    int getX() const { return _x; }
protected:
    int _x;
};
```

To modify the X property of the document while processing either a document view event/action or a document action, it is not recommended to call directly the `setX` method of the document. It is more appropriate to implement a command class (called `ChangeXPropertyCommand` in this example) that will modify this property:

```
class ChangeXPropertyCommand
: public IlvDvCommand
{
    ChangeXPropertyCommand(MyDocument* document, int newX)
        : _document (document),
          _newX (newX)
    {
```

```

        _oldX = document->getX();
    }
    virtual void doIt() { setX(_newX); }
    virtual void undo() { setX(_oldX); }
    void setX(int x) { _document->setX(x); }
protected:
    MyDocument* _document;
    int _newX;
    int _oldX;
};

```

Therefore, the implementation of a view or the document itself should invoke the following code to change the X property (instead of calling directly the `setX` method of the document):

```
document->doCommand(new ChangeXPropertyCommand(document, newX));
```

This code will execute the `ChangeXPropertyCommand` command by calling its `doIt` method, and will store it within a command history internally managed by the document.

Use the following method of the `IlvDvDocument` class to manage commands:

```
void doCommand(IlvDvCommand* cmd,
              IlvBoolean updateUI = IlvTrue,
              IlvBoolean bSetModified = IlvTrue);
```

This method is called to add the command object `cmd` to the history of internal commands. Then, the command is executed by calling its `IlvDvCommand::doIt` method. The `updateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated. The `bSetModified` parameter specifies whether the modification flag of the document must be set to true.

Undo / Redo / Repeat Actions

The Undo, Redo, and Repeat actions are automatically managed by the document. To process these actions, the document invokes the following methods (which can be overridden for specific uses):

- ◆ `virtual IlvBoolean canUndo() const;`

This method returns true if the command that has just been executed can be undone. If there is no command that can be undone, for example if the document has not been modified, this method returns false.

- ◆ `virtual void undo(IlvBoolean bUpdateUI = IlvTrue);`

This method calls the `undo` method of the last command executed. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

- ◆ `virtual IlvBoolean canRedo() const;`

This method returns true if the command that has just been undone can be redone by calling the `IlvDvCommand::doIt` method. If there is no command that can be done, for example if the document has not been modified, this method returns false.

- ◆ `virtual void redo(IlvBoolean bUpdateUI = IlvTrue);`

This method calls the `doIt` method of the last undone command. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

- ◆ `virtual void repeat(IlvBoolean bUpdateUI = IlvTrue);`

This method repeats the last command executed. This command is copied by calling its `IlvDvCommand::copy` method. The copy is added to the commands history and then executed. The `bUpdateUI` parameter specifies that the UI of the Undo, Redo, and Repeat commands must be updated.

Reflecting Changes Made In the Data to Associated Views

You have seen how to modify the data of a document by using commands. However, you still need to see how to notify the views associated with the document to reflect these changes.

A document can have several views of different types. Therefore, to communicate with its views, a document sends generic messages that are interpreted by each view depending on their type. To send generic messages to its views, a document uses the following method:

```
void notifyViews(const char* messageName,
                IlvDvDocViewInterface* exceptView, ...);
```

- ◆ The name of the message is `messageName`. It must not be the name of an action (such as Copy, OpenDocument, and so on).
- ◆ The `exceptView` parameter specifies a view that must not be notified. The value of this parameter is usually 0. It can also be the view returned by the call to `getCurrentCallerView()` (this method returns the view that is currently notifying the document of an event). In this case, you may want this view *not* to be notified because it may have already modified its contents before it notified the document.
- ◆ The variable list of parameters that follows depends on the message name. For example, if a document contains information on an employee and that a command has just changed the name of that employee, the document notifies its views of this change as follows:

```
void EmployeeDocument::changeName(const char* name)
{
    _employeeName = name;
    notifyViews("NameChanged", 0 /* Notify all views */, name);
}
```


To receive a message, a view (or any other class that implements the `IlvDvInterface` interface) must specify an entry in its interface declaration. This entry makes a message name and its parameters correspond with a method of the class. This is explained in more detail in Chapter 4, *Application Framework Interfaces*.

The sample can now be completed so that the view of an employee document can receive the message `NameChanged`:

```
IlvDvBeginInterface(EmployeeView)
/* The message "NameChanged" with one parameter const char* name
   is processed by the method:
   EmployeeView::nameChanged with one parameter const char* name */

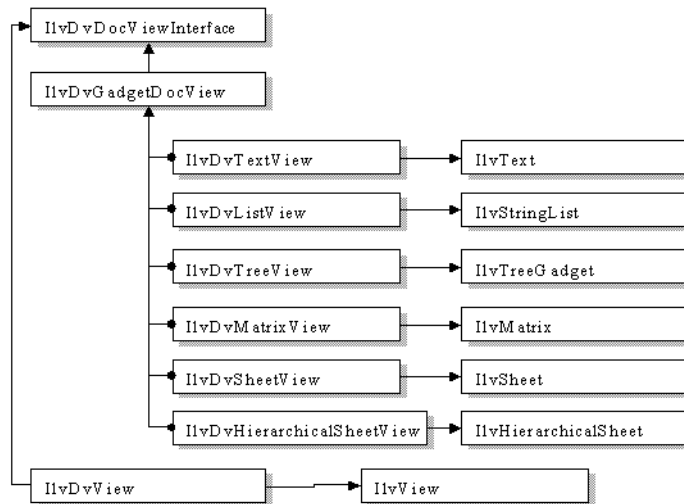
   Method1(NameChanged, nameChanged, const char*, name)
IlvDvEndInterface1 (IlvDvFormView)

/* The nameChanged method is automatically called when an EmployeeDocument
   notifies its views giving the message name "NameChanged" */

void EmployeeView::nameChanged(const char* name)
{
    IlvTextField* nameField = getEmployeeNameField();
    nameField->setLabel(name, IlvTrue);
}
```

Implementation of a Document View Class

A document view class is derived from the `IlvDvDocViewInterface` class. It shows the contents of its associated document and allows the end user to edit it. Here is the inheritance tree of the `IlvDvDocViewInterface` class:



When deriving from this class, only the `IlvDvDocViewInterface::initializeView` method needs to be overridden.

```
virtual void initializeView();
```

This method is called to initialize the document view object according to the document data. For example, a list view can be filled according to a data set stored in the document. A sample of the body of this method is shown here:

```
void
ListView::initializeView()
{
    IlvDvListView::initializeView();
    ListDocument* document = getListDocument();
    IlvUInt count;
    Element* const* elements = document->getElements(count);
    for(IlvUInt iElement = 0; iElement < count; iElement++)
        addString(elements[iElement]->getName());
}
```

Interactions

You have seen how a document view can show the contents of its document.

However, you will also need to edit the contents of a document by interacting with a view. You will want interactions made on the view to be translated into changes in the document data.

Reminder: *It is recommended to use Application Framework commands to do this.*

Consider a list view that lets the user edit a list of names stored within the document associated with the view. You want the user to be able to remove a name (the selected name in the view) by pressing the Del key. To do so, proceed as follows:

```
// The list view tracks the event and makes the changes to the document
// through a command
void ListView::handleGadgetEvent(IlvEvent& event)
{
    if (event.type() == IlvKeyDown) {
        IlvUShort c = event.data();
        if (c == IlvDeleteKey) {
            getNamesDocument()->doCommand(new RemoveNameCommand(getNamesDocument(),
                                                                getSelectedName());
            return IlvTrue;
        }
    }
    return IlvStringList::handleGadgetEvent(event);
}

// Here is the implementation of the command class
class RemoveNameCommand
: public IlvDvCommand
{
public:
    RemoveNameCommand(NamesDocument* document, const char* name):
        _document(document), _name(name) {}
    virtual void doIt() { _document->removeName((const char*)_name); }
    virtual void undo() { _document->insertName((const char*)_name); }
protected:
    NamesDocument* _document;
    IlString _name;
};
```

This showed how events that occur on a view can be reflected to a view through the use of commands. However, the view still has to be refreshed to reflect this change.

In this sample, the selected name item still has to be removed from the list when the user presses the Del key. To do this, the document notifies its associated views when it removes a name from its list of names. To communicate with its views, the document sends generic messages to its associated views, as shown in section Commands.

The sample will now be completed:

```
// The document notifies its views when it removes a name from its
// list of names
void NamesDocument::removeName(const char* name)
{
    _namesArray.removeName(name);
    notifyViews("RemoveName", 0, name);
}

// The list view updates its list when the document notifies it that it
// has removed a name from its list. First, the list view class associates
// the RemoveName message with its removeName method. Thus, this method will
// be called when the user notifies its views with the RemoveName message.
```

```
IlvDvBeginInterface(ListView)
Method1(RemoveName, removeName, const char*, name);
IlvDvEndInterface1(IlvDvListView)
void ListView::removeName(const char* name)
{
    IlShort index = getPosition(name);
    if (index != (IlShort)-1) {
        remove(index);
        redraw();
    }
}
```

For more information on managing events in document views, see `samples manager` and `synedit`, both provided in the `samples` directory.

Application Framework Interfaces

This chapter describes how to use the Application Framework interface. It is divided as follows:

- ◆ *The Interface Mechanism*
- ◆ *Declaring an Interface*
- ◆ *Naming Convention for Macros*

The Interface Mechanism

Application Framework provides an interface mechanism that allows you to:

- ◆ Track and process GUI actions (see the chapter on *Actions*).
- ◆ Perform introspection on your classes.
- ◆ Script your classes.

This interface mechanism associates a name with a method or field of a class. The name of this method or field depends on how the interface mechanism is used. For introspection and scripting, the name is a key that identifies the method.

Declaring an Interface

Here is a small sample showing how to declare an interface to a class (called 'A'):

```
// Declaration of class A
class A
: public IlvDvInterface
{
public:
    void setX(int x) { _x = x; } < /FONT >
    int getX() const { return _x; }

protected:
    int _x;
};

// Implementation file of A. Use the following macros to
// introspect methods setX, getX, and field _x:

IlvDvBeginInterface(A)
    Method1(SetX, setX)
    TypedMethod (GetX, getX)
    Field(X, _x)
IlvDvEndInterface()

....
// Using an instance of A as an interface, it is possible
// to invoke its methods and to modify its field
// without being aware of class A !!!
A* a = new A;
IlvDvInterface* interf = a;

// First we invoke its methods:
IlvDvValue returnedValue;
interf->callMethod(IlvGetSymbol("SetX"), &returnedValue, 100);
interf->callMethod(IlvGetSymbol("GetX"), &returnedValue);
assert((IlvInt) returnedValue == 100);

// Then, we modify its field directly:
interf->setFieldValue(IlvGetSymbol("X"), 200);
assert((IlvInt)interf->getFieldValue(IlvGetSymbol("X"),
                                     &returnedValue) == 200);
```

Naming Convention for Macros

This section discusses the naming conventions for macros used for scripting and introspection.

For methods:

- ◆ The root of the macro name must be Method.

- ◆ If the declared method returns a value, the macro must begin with the prefix `Typed`.
- ◆ If the declared method contains arguments, the macro must end with the suffix `[Number of Parameters]`.

For fields:

- ◆ The macro name is `Field`.

Examples are included in Table 4.1:

Table 4.1 *Macro samples*

Methods to “export”	Macro declarations
<code>Violate getPosition() const;</code>	<code>TypedMethod (GetPosition, getPosition, llvFloat)</code>
<code>const char* set(int);</code>	<code>TypedMethod1 (Set, set, int, ExportedFirstParameterName, const char*)</code>
<code>void setPosition(int x, int y);</code>	<code>Method2 (SetPosition, setPosition, int, X, int, Y)</code>

For scripting and introspection, the first macro parameter is used to identify the method or the field given as the second parameter.

For more information on introspection, see the sample dealing with introspection provided in the *samples* directory.

Actions

This chapter describes how to use the action events provided with Application Framework. It has the sections:

- ◆ *Activating an Action Event*
- ◆ *Processing an Action Event*

Activating an Action Event

Using interfaces, Application Framework provides a mechanism that makes it easy to process actions. The application processes the activation of a menu item in a menu or in a toolbar by generating an action event. This action event is generated according to the action associated with the activated menu item.

Then, the action event is sent to the following targets in this order:

- ◆ The active document view, which is the active view inside the active view frame.
- ◆ The document associated with the active document view.
- ◆ The active view frame.
- ◆ The views and their associated documents, which are inserted into dockable bars.
- ◆ The main window.

- ◆ The action processors declared to the application.
- ◆ The application itself.

Processing an Action Event

To process an action event, a class must insert an `Action` macro in its interface. The first parameter of the `Action` macro is the name of the action, and the second parameter is the name of the method that will process this action.

For example, here is a text view that manages the `Cut` action event:

```
IlvDvBeginInterface(MyTextView)
    Action(Cut, myCut)
IlvDvEndInterface1(IlvDvTextView)

void
MyTextView::myCut()
{
    ...
}
```

A document or a view can manage the action state the same way as it processes an action event. It does this using the macro `RefreshAction([ActionName], [MethodName])`.

For example, here is a text view that manages the `Cut` action state:

```
IlvDvBeginInterface(MyTextView)
    RefreshAction(Cut, refreshCut)
IlvDvEndInterface1(IlvDvTextView)

void
MyTextView::refreshCut(IlvDvActionDescriptor* desc)
{
    desc->setValid(isRelevantSelection());
}
```

The `refreshCut` method will be called each time the document (and its associated document views) becomes active. Since this may not be sufficient, it is possible to force the checking of the action state by calling the application method `refreshAction([ActionName])`. In the previous sample, `refreshAction(IlvGetSymbol("Cut"))` can be invoked, for example, each time the selection changes in the text view.

Index

Symbols

.odv files **19**
.spj files **44**

A

action
 creating **32**
action event
 processing **66**
action parameters
 setting **28**
action state
 forcing **66**
actions
 description **65**
adding
 menu item **20**
 popup menu item **35**
 popup submenus **35**
 toolbar item **21**
Application Framework
 code **48**
 inheritance tree **51**
 overview **9, 65**
Application Framework Editor
 creating an application **17**
 description **10**
 develop an application **17**

 document type **18**
 main window **14**
 menu **44**
 startup **13**
 toolbar **16, 44**
 using **13**
 workspace **16**
application name **20**
application parameters
 setting **19**
applications
 generating **44**
 generating data **44**
 generating specific files **44**
 opening **14**

B

betweenQuotes **52**

C

C++
 prerequisites **7**
cascade document views **45**
closing
 Application Framework Editor **45**
 ODV file **44**
components palette **15**
creating

- action **32**
- application **17**
- menu items **20**
- options file **19**
- popup menu **34**
- popup menu items **35**
- popup submenu **35**
- toolbar items **21**
- custom generation **43**
- customize tool **44**

D

- data
 - generating **44**
- data parameters
 - setting **39**
- developing
 - application **17**
- dialog parameters
 - setting **35**
- display next window **45**
- display previous window **45**
- document
 - description **11**
 - general parameters **22**
 - setting selected parameters **24**
 - toolbar parameters **28**
- document parameters
 - setting **21**
- document type **18**
- Document/View architecture
 - classes **48**
 - description **10**

E

- exiting **45**

F

- file
 - new **44**
 - operations menu **44**
- forcing

- action state **66**
- frame window **10**

G

- generate all **43, 44**
- generate data **43**
- generating
 - all **43**
 - application **44**
 - custom **43, 44**
 - data **43, 44**
 - parameters **40**
 - specific files **44**
- generation
 - all **43**
 - custom **43**
 - data **43**
 - operations menu **44**
 - setting project parameters **40, 44**
- generic document **19**
- grapher application **19**
- GUI events
 - track and process **10, 61**

H

- handling menu and toolbar items **50**

I

- IlvApplication class **50**
- IlvDvApplication class **48, 50**
 - description **48**
 - setAppOptionsFilename member function **49**
 - setUserOptionsFilename member function **49**
- IlvDvDocument class **48**
 - description **48**
 - initializeDocument member function **51**
- IlvDvDocViewInterface class **56**
 - description **48**
 - initializeView member function **57**
- IlvDvSerializable class **52**
- IlvDvStream class **52**
- implementation

- new document **51**
- serialization **51**
- inheritance tree **51**
 - document view class **57**
- initializeDocument member function
 - IlvDvDocument class **51**
- initializeView member function
 - IlvDvDocViewInterface class **57**
- interactions **57**
- interface declaration
 - sample **62**
- interface mechanism **61**
- introspection **61**
- istream **52**

L

- loading
 - ODV file **44**
- loading data **52**

M

- macros
 - action **66**
 - naming conventions **62**
 - RefreshAction **66**
- main file
 - description **50**
 - sample code **50**
- Main window title **20**
- manager application **19**
- managing options data **50**
- manual
 - naming conventions **8**
 - notation **8**
- menu items
 - adding **20**
- menus
 - creating popup **34**
 - creating popup submenus **35**
- modules
 - insert and remove **44**

N

- naming conventions **8**
 - examples **63**
 - fields **63**
 - macros **62**
 - methods **62**
- new application **14**
- new project **44**
- notation **8**

O

- ODV file
 - closing **44**
 - loading **44**
 - saving **44**
- opening
 - application **14**
 - ODV file **44**
- option files
 - application **49**
 - Application Framework **49**
 - description **49**
 - user **49**
- options file **19**
- ostream **52**

P

- palette **15**
- parameters
 - action **28**
 - application **19**
 - data **39**
 - dialog **35**
 - document **21**
 - general document **22**
 - generating **40**
 - popup **32**
 - selected document **24**
 - toolbar for a document **28**
 - window **25**
- popup
 - creating menu **34**

- creating submenu **35**
- setting parameters **32**

- popup menu items
 - adding **35**

- processing
 - GUI events **61**

- profile options file
 - Unix **49**

- Windows **49**

- project application **19**

Q

- quitting

 - Application Framework Editor **45**

- quotation marks **52**

S

- saving

 - ODV file **44**

- saving as

 - ODV file **44**

- saving data **52**

- script project **44**

- script project file **44**

- setAppOptionsFilename member function

 - IlvDvApplication class **49**

- setting

 - action parameters **28**

 - application parameters **19**

 - data parameters **39**

 - dialog parameters **35**

 - document parameters **21**

 - document toolbar parameters **28**

 - general document parameters **22**

 - popup parameters **32**

 - project generation parameters **40, 44**

 - selected document parameters **24**

 - window parameters **25**

- setUserOptionsFilename member function

 - IlvDvApplication class **49**

- starting

 - Application Framework Editor **13**

 - new window **45**

- strings

 - blank spaces **52**

T

- text application **19**

- tiling

 - windows horizontally **45**

 - windows vertically **45**

- toolbar

 - Application Framework Editor **16**

- toolbar items

 - adding **21**

- tools

 - menu **44**

- tracking

 - GUI events **61**

W

- window

 - display next **45**

 - display previous **45**

 - menu **44**

- window parameters

 - setting **25**

- windows

 - cascade **45**

 - starting new **45**

 - tiling horizontally **45**

 - tiling vertically **45**

- workspace **16**