UIM/X

User's Guide

Integrated Computer Solutions Incorporated

Integrated Computer Solutions, Inc.

54 Middlesex Turnpike, Bedford, MA 01730

Tel: 617.621.0060

Fax: 617.621.9555

E-mail: info@ics.com

WWW: http://www.ics.com

UIM/X Trademarks

Contents

Preface

Overview

Well-designed applications with iconic user interfaces have many advantages: they are easy to learn, easy to use, and can provide users with the support needed to effectively work with the application. Unfortunately, good iconic user interfaces are difficult and time-consuming to develop.

UIM/X allows you, the software developer, to interactively create and test sophisticated iconic user interfaces quickly and easily. You can use UIM/X for a wide range of applications—from existing applications currently driven by keyboard input to state-of-the-art applications with direct-manipulation user interfaces. UIM/X is a powerful tool that can help you create iconic user interfaces in a fraction of the time it normally takes to program. You don’t have to be a user interface specialist to use UIM/X—virtually any software developer can use it.

UIM/X brings you all the benefits of iconic user interfaces, none of the drawbacks, and many additional benefits exclusive to it.

Who Should Use this Guide

This manual assumes you are familiar with the basics of UIM/X. Before using this manual, review the *UIM/X Beginner's Guide*.

This manual also assumes that you have some knowledge of programming and a general understanding of the X Window System. You should also know how to use common items such as menus, buttons, and scroll bars. If you are not familiar with these items, you may find it useful to review the *OSF/Motif User's Guide* and the *UIM/X Motif Developer's Guide*.

Before you begin, check with your system administrator to ensure that the software has been installed as described in the *UIM/X Installation Guide*.

Before You Read this Guide

This guide makes the following assumptions:

- You are familiar with the basic functions of selecting from menus and dialog boxes; opening, moving, resizing and closing windows, and clicking icons.
- You understand the functions of the three mouse buttons, which this guide refers to as the Select button (left button), the Adjust button (middle), and the Menu button (right). See "Using the Mouse" on page xiii.
- Either you have enough familiarity with programming to enter your own callback code; or you are using the novice mode of UIM/X to help design user interfaces for which a colleague can provide any code required.

The UIM/X Document Set and Related Books

This section lists the UIM/X document set, and provides a suggested list for further reading.

The following list is the complete UIM/X document set:

- *UIM/X Installation Guide*. Explains how to install and run UIM/X. Includes information on the files provided with UIM/X, backwards compatibility issues, and compiler considerations.
- *UIM/X Beginner's Guide*. Introduces UIM/X by presenting Novice Mode, the simplified Palette that enables new users to be productive immediately. Includes information on a number of important features for creating, testing and running applications.
- *UIM/X Tutorial Guide***.** A series of step-by-step tutorials, teaching tools and techniques that will greatly assist you in developing your own applications. Features tutorials in Novice Mode, Standard Mode, and on advanced topics.
- *UIM/X User's Guide*. Explores the UIM/X features common to both Motif and cross-platform development. Includes discussions of how to use UIM/X's editors to set properties, add behavior, etc.
- *UIM/X Motif Developer's Guide*. An in-depth guide to the widgets, features and capabilities of UIM/X as they relate specifically to Motif development.
- *UIM/X Advanced Topics***.** Describes how to customize UIM/X, including integrating new widget and component classes into the executable. Includes reference information of an advanced technical nature.
- *UIM/X Reference Manual*. A comprehensive list of properties, methods, and events, plus more, for Motif development. Designed for the experienced developer.

Suggested Reading

For more information on designing GUIs, see any of the following books:

- *OSF/Motif Style Guide release 1.2*
  (Prentice Hall, 1993, ISBN 0-13-643123-2)
- *Visual Design with OSF/Motif*
  (by Shiz Kobara, Addison-Wesley, 1991, ISBN 0-201-56320-7)
- *New Windows Interface: An Application Guide*
  (Microsoft Corporation, 1994, ISBN 1-55615-679-0)
- *Human Interface Guidelines: The Apple Desktop Interface*
  (Addison-Wesley, 1987, ISBN 0-201-17753-6)

How this Guide Is Organized

Chapter 1, "Running UIM/X" , covers the basic concepts of starting and running UIM/X.

Chapter 2, "Building Interfaces", explains some of the features that allow you to interact with UIM/X to build your interface.

Chapter 3, "Building Palettes", explains how to build and modify palettes.

Chapter 4, "Setting Properties", explains how to use the Property Editor to manipulate the appearance and behavior of the objects in your interface.

Chapter 5, "Specifying Callbacks and Connections", shows how to use the Callback Editor and the Connection Editor to specify behavior.

Chapter 6, "Editing Interface Code", explains how to use the Declaration Editor to edit portions of your generated code.

Chapter 7, "Building Parametric Interfaces", explains how to build interfaces with a dynamic initial state.

Chapter 8, "Building Reusable Interface Components", shows how to save time and increase consistency by turning a useful group of widgets into a reusable component.

Chapter 9, "Working with Components, Subclasses, and Instances", explains how to better use components by creating subclasses and instances.

Chapter 10, "Methods", describes the Method Editor and explains how to use it while building your interfaces.

Chapter 11, "Using the Interpreter", explains how to use the Interpreter to evaluate and test your code.

Chapter 12, "Mixing Compiled and Interpreted Code", shows how to augment the UIM/X executable with the object code of other applications.

Appendix A, "Advanced C++ Programming in UIM/X", explains object-oriented development, and how to use C++ within UIM/X.

Appendix B, "Frequently Asked Questions", provides answers to a list of frequently asked questions.

Some Terms You Should Know

Certain basic terms recur throughout this guide, and it helps to understand them from the outset.

An *object* is a building block you can use to build an interface with UIM/X.

A *Motif widget* is an object whose appearance and behavior precisely follows the *OSF/Motif Style Guide*. The novice mode of UIM/X supports a number of popular Motif widgets, including Push Button, Label, Text Field, and more.

A compound object consists of several Motif widgets combined into one object for your convenience. The novice mode of UIM/X supports a number of compound objects, which save you the time you might otherwise spend creating them, including Application Window and Group Box.

An *interface* is a window or dialog box that you build up from objects with UIM/X. The novice mode of UIM/X supports four different types of interfaces: Application Window, Secondary Window, Message dialog box, and File Selection dialog box. Certain menu options refer to an interface, such as Save Interface; these act only on your selected interface.

A *project* contains all the interfaces (i.e., windows and dialog boxes) and their associated files for a certain GUI you are building with UIM/X. The program can automatically save and generate code for an entire project in one step. Certain menu options refer to a project, such as Save Project; these act on all the windows and dialog boxes in your project.

Conventions Used in this Guide

Installation Directories

Product installation directories can depend on the platform or the user's preferences. To keep things simple, this guide uses general names for product installation directories. The following table lists the name and the corresponding product installation directory:

Name	Description
uimx_directory	The UIM/X installation directory.

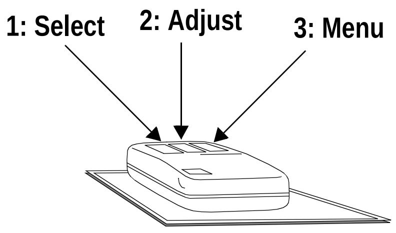
Typographic Conventions

The following table describes the typographic conventions used in this guide.

Typeface or Symbol	Meaning	Example
`AaBbCc12`	The names of commands, files, and directories; or onscreen output; or user input.	Edit your `.login` file. `%You have mail.` Use `ls -a` to list all the files.
AaBbCc12	A placeholder you replace with your actual value; or words to be emphasized; or book titles.	To delete a file, type `rm` *filename.* You *must* be `root` to do this. See Chapter 6 in the *User's Guide*.
File⇒Open	The Open option in the File menu.	Choose the File⇒Open command.
Alt+F4	Press both Alt and F4 at once.	Press Alt+F4 to exit.
Return	The key on your keyboard marked Enter, Return, or ↵.	Press Return.

Using the Mouse

Before starting the tutorial, take a moment to review the location and usage of your mouse buttons, as illustrated below and in the following table:



Button	Called	Is used for
1	Select	Selecting objects, menus, toggles, and options.
2	Adjust	Resizing and moving objects.
3	Menu	Displaying popup menus.

Throughout this book, you use the mouse buttons along with the mouse pointer to make selections, move the input pointer, or position the text insertion point. You can perform any of the following mouse operations.

Operation	Description
Point to	Move the mouse to make the pointer go as directed.
Press	Hold down a mouse button.
Release	Release a mouse button after pressing it.
Click	Quickly press and release a mouse button without moving the mouse.
Drag	Move the mouse while pressing a mouse button.
Double-click	Click a mouse button twice in rapid succession without moving the mouse pointer.
Triple-click	Click a mouse button three times in rapid succession without moving the mouse pointer.

In general, instructions for mouse operations include the name of the mouse button. The exceptions are Click, Double-click, and Drag. These common operations may be described without specifying a mouse button. For example:

- Click on the `applWindow1` icon in the Interfaces Area of the Project Window.
- Drag the Push Button icon from the Palette.

In these cases, use the Select button to click and double-click, and the Adjust button to drag.

Setting Application Defaults

Application Defaults configure the way UIM/X looks and set the default preferences for many of its operations. You can set the Application Defaults for all UIM/X users or for a single user. For more details on setting your Application Defaults see *UIM/X Advanced Topics*.

For optimum performance, set the following resources in your Application Defaults.

```
Mwm*autoKeyFocus: false
Mwm*clientAutoPlace: false
Mwm*focusAutoRaise: false
Mwm*focusFollowsPointer: true
Mwm*keyboardFocusPolicy: pointer
```

Note: The resources above prefixed with `Mwm` are specific to the Motif Window Manager. If you are using a different window manager consult your Systems Administrator for the equivalent settings.

Running UIM/X

1

Overview

This chapter covers the basic concepts of running UIM/X. Topics covered in this chapter include:

- Starting UIM/X
- Using UIM/X command-line options
- Starting UIM/X in Novice Mode
- Configuring the start-up desktop
- Resetting UIM/X
- Exiting UIM/X

Starting UIM/X

To Start a UIM/X Session

1. Start the X Window System.
2. Open a terminal window.
3. Start UIM/X from the UNIX prompt:

```
        uimx &
```

   If your `PATH` variable does not provide the full path to the UIM/X executable, you have to specify it when you run UIM/X:

```
        uimx_directory/bin/uimx &
```

4. After a brief pause, a copyright notice window appears on the screen, to show that UIM/X is being initialized. Click on the OK to make the notice disappear, or just wait—the notice will disappear once UIM/X is initialized. Clicking on Cancel *quits* UIM/X.
5. When you see the UIM/X Project Window and the Ux palette (see Figure 1-1), you are ready to begin.
6. Iconify the terminal window.

The Project Window is the main window of UIM/X. When you start UIM/X, an empty Project Window appears on your screen. The Ux palette, from which you can select the objects for your project, appears beside the project window as shown in Figure 1-1.

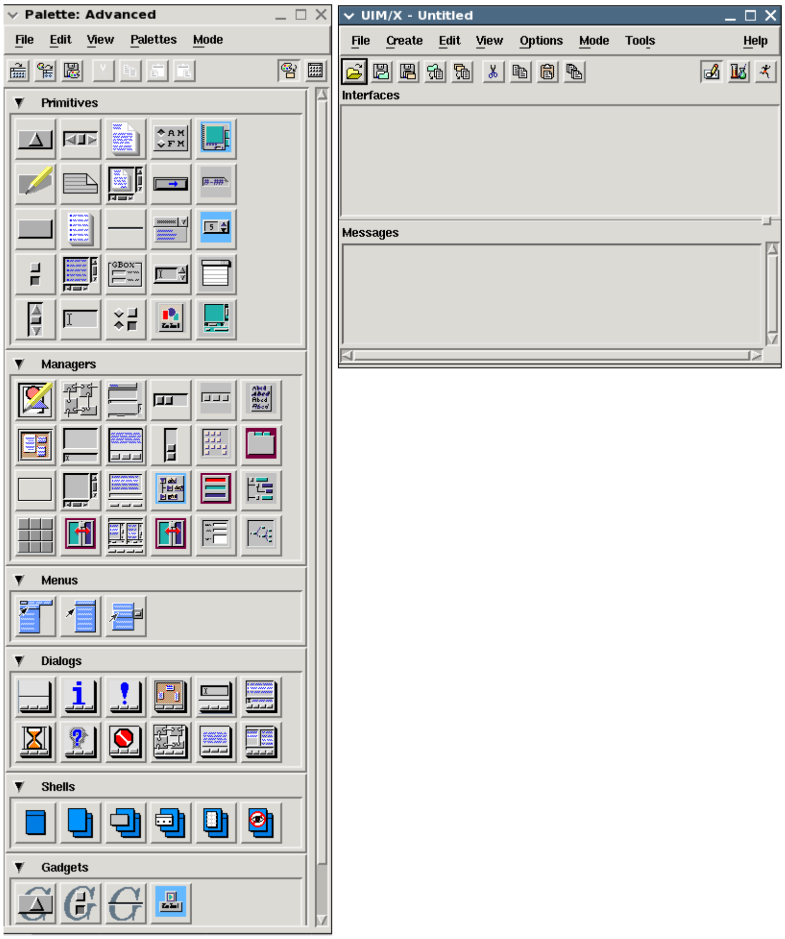


Figure 1-1 Project Window and Ux Palette

Using UIM/X Command-Line Options

By default UIM/X starts up in Design mode with the Ux palette and an empty project window. By specifying options on the command line, however, you can start UIM/X and load projects, interfaces, and palettes.

UIM/X accepts a number of command-line options. These include:

`-file` *filename*	*filename* can be a project, an interface, or palette file. The *filename* can include an absolute path name or can be relative to either the current directory or the -dir value.
`-dir` *path*	Set UIM/X's current directory to *path*.
`-novice`	Starts UIM/X in Novice Mode.
`-xrm` *options*	Any specification that you would otherwise put into a resource file. For example: `uimx -xrm "*UxPECompound.set: false"`
`-language` *language*	One of: krc, ansic, or c++

Loading Files

UIM/X supports three basic files types:

.prj	Project file
.i	Interface file
.pal	Palette file

By using UIM/X's `-file` command-line option, you can tell UIM/X to start-up and automatically load a specific project, interface, or palette file. For example, the following command will cause UIM/X to load the Toolbar project at start up.

```
    uimx_directory/bin/uimx -file ToolBar.prj &
```

You can load an interface and palettes the same way.

Starting UIM/X in Novice Mode

UIM/X provides a Novice Mode to help new users learn how to use the product. When you start UIM/X in Novice Mode, you get a basic set of features and tools that makes it easier to learn UIM/X.

To start UIM/X in Novice Mode, specify the `-novice` option on the UIM/X command line:

```
uimx_directory/bin/uimx -novice &
```

Figure 1-2 shows the start-up interface for Novice Mode. For more information about Novice Mode, see the *UIM/X Beginner's Guide*.

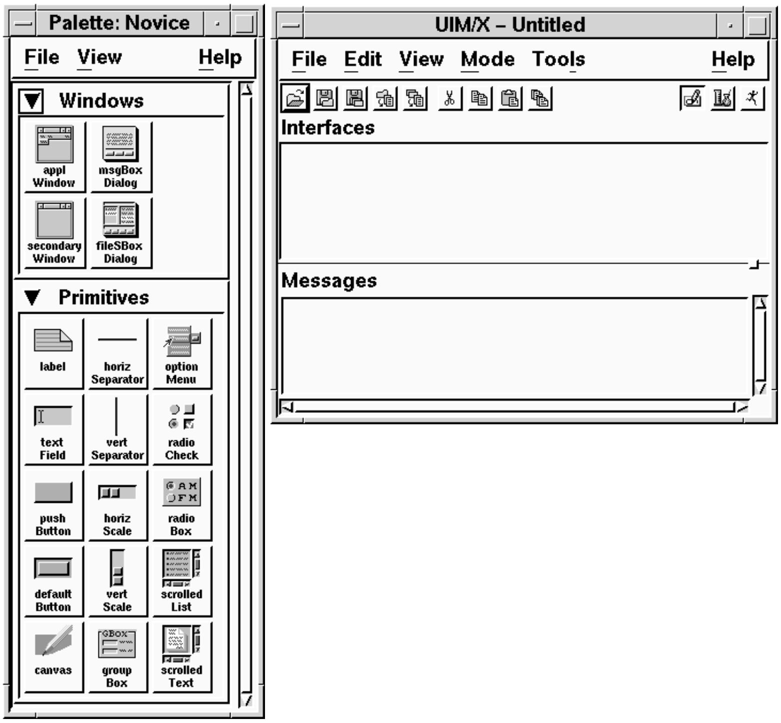


Figure 1-2 UIM/X Novice Mode

Starting UIM/X in a Different Language Mode

By default, UIM/X starts in C++ mode. Alternatively, you can start UIM/X in ANSI C mode or K&R C mode by using the `-language` argument.

For example, to start UIM/X in K&R C mode, specify the `-language` option on the Unix command line, as follows:

```
uimx_directory/bin/uimx -language krc &
```

Configuring UIM/X's Start-Up Desktop

Note: In general, UIM/X will provide the best viewing results when displayed on a color monitor with a minimum resolution of 1280 x 1024 pixels and suitable color support (>256 colors) for typical X applications.

The start-up desktop is the initial working environment UIM/X provides for the user at start-up. You can configure the following elements of UIM/X's start-up desktop:

- The start-up mode. UIM/X can be started in either Design or Test Mode.
- UIM/X's main application window, referred to as the start-up interface. You can use either the Project Window or the Browser as the start-up interface.
- The additional tools and editors that pop-up at start-up. The Property Editor and the Browser can be added to the start-up desktop.
- The initial arrangement and size of the interfaces in the start-up desktop.
- The interfaces, project, and palettes loaded at start-up.

Configuring the start-up desktop allows you to tailor UIM/X to the needs of your users. Resources control the configuration of the start-up desktop. You can set these resources in the Application Defaults. For example, the following resource specification makes the Browser the start-up interface:

```
Uimx3_0*UxStartupInterface.value: browser
```

You can hard-code resource settings. For example, you could also create a version of UIM/X that always starts in Test mode by hard-coding the value of `UxStartupInterface.value`.

The following table is provided as a quick reference. It lists the resources and gives a summary description of the possible resource settings. The remainder of this section discusses the configurable elements of the start-up desktop in detail.

Resource	**Value**	**Description**
`UxStartup Interface.value`	`project`	Project Window is start-up interface.
	`browser`	Browser is start-up interface.
	`test`	Load interface or project and enter Test mode.
`UxBrOnStartup.set (applicable only with the Project Window as the start-up interface)`	`false`	No Browser in start-up desktop.
	`true`	Browser pops up at start-up.
`UxPEOnStartup.set`	`false`	No Property Editor at start-up.
	`true`	Property Editor pops up at start-up.
`UxBrowserVisible.set (applicable only with the Browser as the start-up interface)`	`false`	Browser Area not visible.
	`true`	Browser Area visible.
`UxPaletteAreaVisible. set`	`false`	Palettes Area not visible.
	`true`	Palettes Area visible.
`UxStartingPalettes.va lue`	`text`	List of `.pal` files to load at start-up.
`UxPalettePath.value`	`text`	Path to `.pal` files.
`brgeometry`	`geometry string`	Browser size and position.
`BrOutlineWindowHeight (applicable only with the Browser as the start-up interface)`	`pixel`	Height of the Browser Area (the window where the object hierarchy is displayed).
`BrMessageWindowHeight`	`pixel`	Height of the Browser Messages area.

Resource	Value	Description
`pegeometry`	`geometry string`	Property Editor size and position.
`prgeometry`	`geometry string`	Project Window size and position.
`PjInterfaceWindowHeight`	`pixel`	The height of the Project Window's Interfaces Area.
`PjPaletteWindowHeight`	`pixel`	The height of the Project Window's Palettes Area.
`PjMessageWindowHeight (applicable only with the Project Window as the start-up interface)`	`pixel`	The height of the Project Window's Messages Area.

Loading Interfaces and Projects

UIM/X can load interface and project files specified on its command line. You tell UIM/X which file to load using the following syntax:

```
uimx [-dir path] [-file filename]
```

where path is the path to the file you want to load, and *filename* is the name of the interface or project file to load.

UIM/X exits if you do not supply a valid file name and path.

You can use these command-line options to load a file into any of the start-up interfaces (Test, Project, or Browser).

Running UIM/X with the Project Window Start-Up Interface

You make the Project Window the start-up interface by setting `UxStartupInterface.value` to `project`:

```
Uimx3_0*UxStartupInterface.value: project
```

This is the default setting found in the UIM/X resource file. When `UxStartupInterface.value` is set to `project`, UIM/X starts in Design mode, and the Project Window is the main application window.

Adding the Browser to the Start-Up Desktop

Setting `UxBrOnStartup.set` to `true` adds the Browser to the start-up desktop when the Project Window is the start-up interface. The Browser will pop-up automatically at start-up.

`UxBrOnStartup.set` is ignored if `UxStartupInterface.value` is not set to `project`.

Running UIM/X with the Browser Start-Up Interface

By default, UIM/X starts up with the Project Window interface and the Ux palette. However, by setting a resource in your Application Defaults, you can cause UIM/X to start with the Browser as the start-up interface. For example, the following resource specifications make the Browser the start-up interface:

```
Uimx3_0*UxStartupInterface.value: browser
Uimx3_0*UxBrowserVisible.set: true
```

The `UxBrowserVisible.set` resources makes the Browser area visible. By default, the Browser area is not visible in the Browser start-up interface. Figure 1-3 shows the Browser start-up interface.

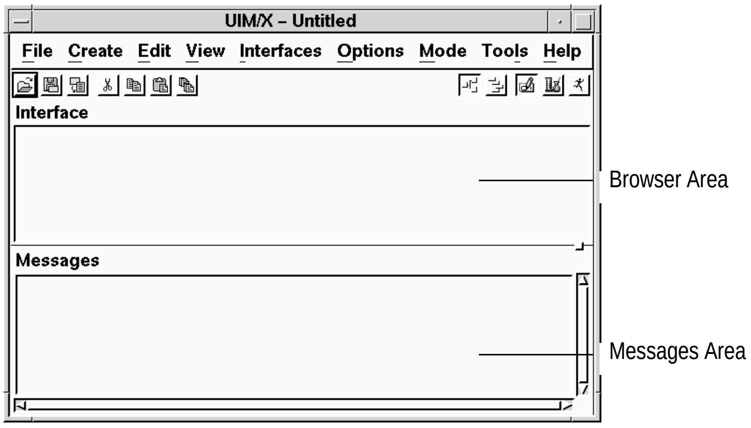


Figure 1-3 Browser Start-Up Interface

If your start-up interface is the Browser, you can still use UIM/X command-line options to load interface and palette files. As a project usually consists of several interface files, specifying a project file at start-up does not load anything into the Browser. An empty Browser is presented, with the available interfaces listed in the Browser Interfaces menu.

During program initialization, UIM/X creates the menu bars and pulldown menus of the Browser and the Project Window. When the Browser is the start-up interface, a number of changes are made to the menu bars and icon bars:

- The Options and Tools menus move from the Project Window menu bar to the Browser menu bar.
- The Browser option is removed from the Project Window's Tools menu.
- An Interfaces menu is added to the Browser menu bar. This menu contains one item for each interface that currently exists within UIM/X. Each time you select an interface from the menu it is loaded into the Browser. The Interfaces menu can list up to 15 interfaces. If there are 16 or more interfaces, the Interfaces menu will include a More item. Selecting this item displays a SelectionBoxDialog for choosing interfaces.
- Open and Open UIL move from the Project Window's File menu to replace Load on the Browser's File menu.
- Exit replaces Close on the Browser's File menu, and Close replaces Exit on the Project Window's File menu.

  These changes reflect the fact that the Browser, not the Project Window, is now UIM/X's main window. An application's main window must permit the user to load files and either reset or exit the application.
- Project Window is added to the Browser's Tools menu.
- Open replaces Load on the Browser's icon bar. The Design, Test, and Run icons move to the Browser.

The client area of the Browser start-up interface also contains two window panes. One window pane is the Browser Area, and the other is the Messages Area. The Project Window has no message area when the Browser is the start-up interface.

When the Browser is UIM/X's main window, there is only one browser window available. Each time you select an interface, the Browser Area displays the object tree (or outline) of the selected interface.

Showing the Palettes Area

The Palettes Area of the Project Window is shown by setting `UxPaletteAreaVisible.set` to `true`. By default, `UxPaletteAreaVisible.set` is set to `false`.

If you make the Palettes Area visible, you will want to adjust the height of the Project Window.

Note also that the UIM/X resource file contains geometry settings for either case.

Loading System Palettes

You can use resources to automatically load palette files at start-up.

Note: As shipped, UIM/X loads a palette of Motif objects at start-up. The UIM/X resource file contains the following resource specifications:

```
Uimx3_0*UxStartingPalettes.value: Ux.pal
Uimx3_0*UxPalettePath.value: uimx_directory/palettes
```

UxStartingPalettes

This resource specifies the palette files loaded at start-up. You can load one or more palette files. These palette files are loaded before any files specified on the UIM/X command line. These palettes are not saved with projects saved by the user.

As shown below, you can use `UxStartingPalettes.value` to specify the absolute pathnames of the palettes you want to load. Note that a pathname must begin with the forward-slash (/) character.

```
Uimx3_0*UxStartingPalettes.value:/usr/bilal/
Bilal.pal
```

Alternatively, you can specify only the names of the palette files in `UxStartingPalettes.value`, and use the `UxPalettePath.value` resource to specify the path to the named files:

```
Uimx3_0*UxStartingPalettes.value: pal1.pal\pal2.pal
Uimx3_0*UxPalettePath.value: /usr/palettes*
```

Note: To load more than one palette, you must insert `\n` between palette file names.

UxPalettePath

This resource specifies where UIM/X will look for the palette files named by the resource `UxStartingPalettes.value`. You must specify an absolute pathname. A pathname must begin with the forward-slash character (/). There should be no trailing spaces after the pathname:

```
Uimx3_0*UxPalettePath.value: /usr/nathalie
```

Setting the Palette Modes

A Palette has two mode properties, `viewMode` and `createMode`. The `viewMode` property determines whether the Palette lists its elements by name, by icon, or by name and icon. The `createMode` property determines whether the Palette is in Create Mode or Edit Mode. These modes are set from the Palette's View and Mode menus.

Suppose you want the Palette loaded at start-up to show only icons and to be in Create Mode. To do this, simply save the Palette with the View and Mode settings you want to see at start-up.

You can also set the initial View and Mode settings of a Palette from within its `.pal` file:

- A Palette's `viewMode` property controls the initial viewing mode of the Palette (by name, by icon, or by name and icon). This property is given an integer value specifying an option on the Palette's View menu:
    - By Name
    - By Icon
    - By Name and Icon
- A Palette's `createMode` property controls whether or not the Palette is in Create Mode when it is loaded. The value `0` sets the initial mode to Create Mode; the value `1` sets the initial mode to Edit Mode.

Adding System Palettes

UIM/X supports two kinds of palettes: user palettes and system palettes. *User palettes*, like interfaces, are part of a project. You load user palettes from the command line with the `-file` option, or by choosing File⇒Open from the Project Window.

System palettes are not part of any project. System palettes are tools provided by UIM/X. The Ux palette is the default system palette. The resource `UxStartingPalettes.value` lists the system palettes loaded by UIM/X at start up.

To add your own palettes to the list of system palettes, set the following resources in your Application Defaults:

```
    Uimx3_0*UxStartingPalettes.value: palette_name
    Uimx3_0*UxPalettePath.value: absolute_path_name
```

The palette name is the name of the palette file with a `.pal` extension. The absolute path name is the path from the root to the directory containing the palettes. For example, the following resource specifications load 3 palettes from the directory `/usr/thien/palettes`:

```
Uimx3_0*UxStartingPalettes.value:
pal1.pal\npal2.pal\npal3.pal
Uimx3_0*UxPalettePath.value: /usr/thien/palettes
```

Setting Start-Up Geometry

You can set the size and position of the UIM/X interfaces in the start-up desktop. With the exception of palettes, the size and position of the interfaces in the start-up desktop are set using standard X Toolkit geometry strings. Palettes have separate `x`, `y`, `width`, and `height` properties.

Note: UIM/X's geometry settings are ignored if either of the OSF/Motif window manager resources `clientAutoPlace` or `interactivePlacement` is set to `true`.

Project Window Geometry

The following sample resource specification shows how to set the initial size and position of the Project Window:

```
Uimx3_0.prgeometry: 500x300+360+1
```

The resources `PjInterfaceWindowHeight`, `PjPaletteWindowHeight`, and `PjMessageWindowHeight` control the height of the windows in the Project Window. Setting a resource fixes the height of the window—the height will not change even if the Project Window is resized. You can override the height setting by using the sash to resize the window. If the resource is not set, the Project Window manages the height of the window.

These resources can be set as follows in a resource file:

```
Uimx3_0*PjInterfaceWindowHeight: 120
Uimx3_0*PjPaletteWindowHeight: 100
Uimx3_0*PjMessageWindowHeight: 120
```

Note: Setting all three of these resources prevents UIM/X from properly resizing the Project Window. If the height of all three windows is fixed, blank space appears when the height of the Project Window exceeds the sum of the window heights.

Browser Geometry

The resource `brgeometry` controls the initial size and position of the Browser. This resource accepts standard geometry strings:

```
Uimx3_0.brgeometry:       500x300+360+330
```

If `brgeometry` contains no `x` and `y` offsets, UIM/X centers the Browser under the current position of the mouse pointer.

The resources `BrOutlineWindowHeight` and `BrMessageWindowHeight` control the height of the windows in the Browser. Setting a resource fixes the height of the window—the height will not change even if the Browser is resized. You can override the height setting by using the sash to resize the window. If the resource is not set, the Browser manages the height of the window.

These resources can be set as follows in a resource file:

```
Uimx3_0*BrOutlineWindowHeight: 280
Uimx3_0*BrMessageWindowHeight: 120
```

Note: Setting both resources prevents UIM/X from properly resizing the Browser. If the height of both windows is fixed, blank space appears when the height of the Browser exceeds the sum of the window heights.

Property Editor Geometry

The resource `pegeometry` controls the initial size and position of the Property Editor. This resource accepts standard geometry strings:

```
Uimx3_0.pegeometry:       600x500-1-1
```

If `pegeometry` contains no x and y offsets, UIM/X centers the Property Editor under the current position of the mouse pointer.

Palette Geometry

A palette's `.pal` file contains its size and position. If you want a palette to have a given size and position when it is loaded:

1. Load the palette into UIM/X.
2. Position and resize the palette.
3. Save the palette.

The Palette's size and position are now stored in the `.pal` file. The next time UIM/X loads the Palette, it sizes and positions the Palette according to the information in the `.pal` file.

Editing the Palette File

You can also set a Palette's geometry by using a text editor to modify the values in the Palette's `.pal` file. A Palette's `x`, `y`, `width`, and `height` attributes control its geometry.

However, if you want UIM/X to check file keys, you still have to load the Palette into UIM/X and save it. Otherwise the modified `.pal` file will not contain the proper file key, and UIM/X will refuse to load the file when file key checking is enabled.

The file key checking feature prevents other users from editing the `.pal` file.

Suppose you had saved a Palette named `dbaseForm`. If you were to examine the file `dbaseForm.pal`, you would see something like the following near the top of the file:

```
*dbaseForm.class: palette
*dbaseForm.name: dbaseForm
*dbaseForm.editable: true
*dbaseForm.x: 472
*dbaseForm.y: 375
*dbaseForm.width: 350
*dbaseForm.height: 206
*dbaseForm.iconBitmap: palette.xpm
*dbaseForm.viewMode: 2
*dbaseForm.createMode: 0
```

You can set the Palette's geometry by modifying the Palette's `x`, `y`, `width`, and `height` properties and saving the file.

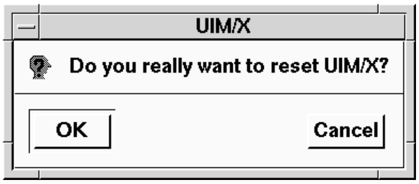
The `viewMode` and `createMode` attributes are discussed in *"Setting the Palette Modes"* on page 12. The `iconBitmap` property is the name of the file containing the pixmap for the Palette's icon.

Resetting UIM/X

When you are working in UIM/X, you may want to start over on a particular project or change and work on another project. You can start over or change projects by resetting UIM/X.

To reset UIM/X

1. Select File⇒Reset from the Project Window to reset UIM/X and begin again from scratch. A dialog box similar to Figure 1-4 appears on the screen:


UIM/X

Do you really want to reset UIM/X?

Figure 1-4 Reset Dialog

2. Click on OK to reset UIM/X. (Click on Cancel to return to UIM/X.)

   Be sure to save any work that you do not want to lose before resetting UIM/X. During the reset process, you will lose any interfaces and Palettes that have not been saved. All current UIM/X windows disappear from view. Once UIM/X is re-initialized, the Project Window and Ux Palette appear on your screen.

You can also reset UIM/X by loading a new project:

1. Select File⇒Open from the Project Window.
2. Select the project you want using the file selection box and click on the OK button.
3. UIM/X will pop-up a warning dialog box advising you that it is going to reset.
4. Click on OK to reset UIM/X and load the new project. (Click on Cancel to return to the current project).

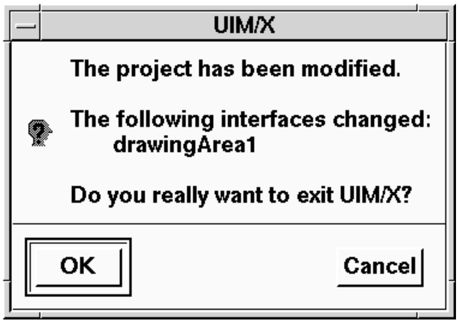
Exiting UIM/X

Before exiting UIM/X, it is important to save your work.

To exit UIM/X:

1. Select File⇒Exit from the Project Window.

   If you attempt to exit UIM/X without first saving your project, a warning message will prompt you to confirm that you wish to exit, as shown in Figure 1-5.


UIM/X

The project has been modified.

The following interfaces changed:
drawingArea1

Do you really want to exit UIM/X?

*Figure 1-5*Warning Dialog

If you do not have any unsaved project, a dialog box appears on the screen, as shown in Figure 1-6:

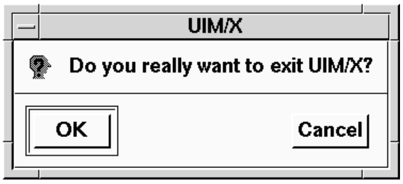


Figure 1-6 Exit Dialog

2. Click on OK to exit UIM/X. (Click on Cancel to return to UIM/X). All UIM/X windows disappear from view.

Note: If you have entered changes in any of its text editors but failed to apply those changes, UIM/X will not warn you that your changes will be lost if you exit UIM/X.

Building Interfaces

2

Overview

UIM/X provides a complete set of tools and features for building interfaces. This chapter explains some of the features that allow you to interact with UIM/X to build your interfaces.

You will learn that you can control how your interface interacts with the window manager, or let UIM/X control the interaction for you. You will learn how to reparent and recreate objects in your interface. You will also learn how to use the Browser to view and edit complex object hierarchies.

Working with Shells

The first object in any object hierarchy is a top-level object. Because a top-level object interacts with the window manager, it must be assigned a Shell object to manage the interaction. In UIM/X, you can create these Shell objects explicitly, or let UIM/X assign them to objects. Shells that you create are referred to as explicit Shells. Shells assigned by UIM/X are referred to as implicit Shells.

By assigning a Shell object explicitly, you can control how the object interacts with the window manager. Through the Property Editor, you have access to all the properties necessary to change the Shell's behavior when it interacts with the window manager.

Implicit Shells, on the other hand, are convenient, but you cannot access their properties.

Implicit Shells

Implicit Shell objects serve two purposes. First, they conveniently enable new users to become productive quickly. You can create interfaces without worrying about setting Shell object properties or how Shells interact with the window manager. As you become more familiar with window manager behavior—and want more control over the interaction—you can reparent your top-level objects. You can assign them explicit Shells more suited to your needs, whose properties you can access.

Components—reusable hierarchies of objects—can also use implicit shells. Objects serving as Components are top-level objects, and must therefore have a Shell. However, an implicit Shell is only temporary, and is automatically stripped away when an Instance of the Component is placed into another object.

Implicit Shells for Dialog Objects

Dialog objects are used to present file selection boxes, warning messages, etc. Because they also interact directly with the window manager, they have a Shell. Dialog objects always receive the same implicit shell, the Dialog Shell.

Unlike other top-level objects with implicit Shells, you cannot reparent a Dialog object to give it another Shell. Even changing the default Shell (see below) does not affect Dialog objects.

Controlling Implicit Shells

While the properties of implicit Shells are not available in the Property Editor, all top-level objects have a property called `AllowShellResize`, to which you have access. This property determines whether or not the Shell grows or shrinks. When set to `false`, the Shell ignores all geometry requests from its children.

No UIM/X swidget exists for implicit Shells. (A swidget is a structure used by UIM/X to maintain state information about the corresponding Motif widget.) Therefore, the Ux Convenience Library functions `UxPut`*Property*`()` and `UxGet`*Property*`()` cannot be applied to implicit Shells. Only Xt or Xm calls can be used.

Changing the Default Implicit Shell

By default, top-level objects receive the implicit Shell topLevelShell. The default Shell is specified through the `defaultShell` resource. You may select another default implicit Shell by changing the resource. Alternatively, you can change the implicit default Shell via the Default Shell selection of the Project Window’s Option menu.

Note: Selecting a new default implicit shell does not change the implicit shells of objects that have already been created.

Changing an Existing Object’s Shell

To Change an Existing Object’s Shell

1. Create the new Shell object.
2. Load the object with the implicit Shell into the Property Editor.
3. Select the Declaration properties from the Category option menu.
4. In the Parent property area, enter the name of the Shell object and click Apply.
5. The Shell is stripped from the object and the object is reparented to the Shell object you created explicitly.

Reparenting Objects

The objects within a hierarchy are related to each other as parent and child. The first object of an interface, the top-level object, has no parent. All child objects have one parent. Some objects cannot have children.

Every object in the hierarchy, except the top-level object, has one and only one parent object and is a child object to that parent. These child objects are usually themselves the parents of still other child objects.

Objects might be reparented for a variety of reasons including:

- The interface design has changed and you want the object to behave differently.
- The object has an implicit shell and you want to reparent to an explicit shell.

Objects can be reparented in three ways:

- Interactively in an interface.
- Interactively in the Browser.
- Using the Property Editor.

Interactive reparenting means dragging and dropping an object in an interface or in an interface hierarchy in the Browser to a valid parent.

The potential parent could reside in the same interface or Browser or in another interface or Browser. However, the operation is mutually exclusive. You cannot drag an object from a Browser and drop it into an interface; you cannot drag an object from an interface and drop it into an object hierarchy in a Browser.

In a receiving interface, the dropped object is reparented to the object directly under the compass when the Adjust mouse button is released. However, if that object cannot accept children, the dropped object is automatically made a child within that object’s hierarchy. The most appropriate parent is automatically selected.

In the Browser, the dropped object must be placed directly over the appropriate parent.

Recreating Objects

Recreating an object during development shows you what the object will look like when created at run time. The Recreate option can be accessed in two ways:

- By choosing Selected Objects⇒Other⇒Recreate.
- By choosing Edit⇒Other⇒Recreate, in both Browser and Project Window.

During development of an interface, sometimes the geometry of an object may not appear to have been properly managed by its parent. For example, if you add a child to an object that already has children, management policies may be updated but not reflected throughout the object hierarchy.

Another common use of the Recreate feature is to verify the position and size of an object. If an object has been moved or resized using the window manager (rather than with the UIM/X move and resize operations), the object will only appear to have changed its geometry. Recreating the object will move it back to its true location and size.

You should use the Recreate command in one of the following situations:

- Before saving the interface as an interface file.
- Before generating code for the interface.
- If you do not get what you expected.
- After using Test Mode.

Naming Conventions

When you create an object, UIM/X automatically gives it a name obtained by appending a number to the object's class name. For example, Form objects you create will be named `form1`, `form2`, `form3`, etc. Interfaces and icons are named similarly.

An object's name is also declared as a variable name in UIM/X, and you can refer to it in code. These variable names are of type "swidget" (for shadow widget). A swidget is a structure used by UIM/X to maintain information about the corresponding Motif widget and to provide a more complete error checking and handling mechanism than Motif.

When using C++ Convenience Library C++ Bindings, Motif objects are declared not as swidgets, but as objects of the appropriate Motif wrapper class provided by the Ux C++ Convenience Library.

You can change the name of any object using the Property Editor. The Name property is located in the Declaration property category. The new name can be any unused, valid variable name.

Note: If you change an object's name after writing your own code that refers to it—in callbacks, for example—be sure to update the code.

Browsing Interfaces

The Browser, shown in , allows you to view and edit complex hierarchies. In the Browser, you can easily reorder and reparent objects. Furthermore, it is a convenient way to select objects and object hierarchies to be stored in a palette for future use.

The Browser displays one interface at a time and its functions are performed on the entire interface or any of its objects.

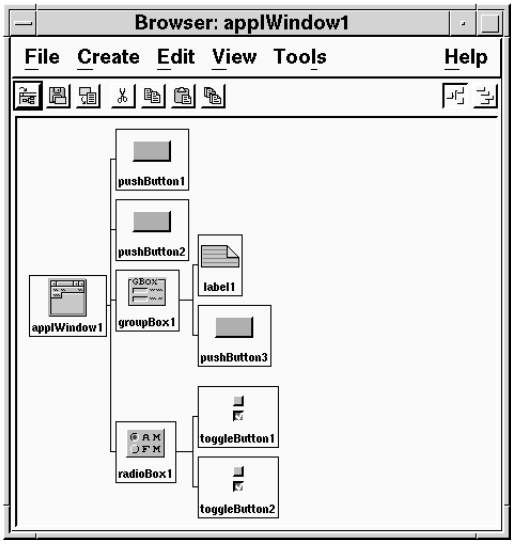


Figure 2-1 Browser

The Browser makes it easy to edit objects, and is very useful when working on objects that are partially or completely hidden. For example, pop-up menu objects are not visible in an interface, but they are visible in the Browser. Using the Browser you can:

- Select objects.
- Duplicate, delete, reparent, move, and resize objects.
- Cut or copy objects to the UIM/X Clipboard, then paste them to any interface or palette.

Opening the Browser

To Open an Empty Browser

Choose Tools⇒Browser from the Project Window, with nothing selected. An empty Browser will appear. An interface can then be loaded by choosing File⇒Load from the Browser or by clicking on the Load icon in the Browser's icon bar.

To Open the Browser with an Interface Loaded

1. Select the interface by clicking on it, or by clicking on its icon in the Interfaces Area of the Project Window.
2. Choose Tools⇒Browser from the Project Window.

OR

1. Select any object in the interface.
2. Choose Selected Objects⇒Tools⇒Browser.

OR

1. Select the interface by clicking on its icon in the Interfaces Area of the Project Window.
2. Choose Selected Interfaces⇒Tools⇒Browser.

The Browser is displayed, loaded with the selected interface.

Clearing and Loading Interfaces

To Clear an Interface from the Browser

Select File⇒Reset from the Browser.

To Load an Interface into the Browser

1. Select the interface to be loaded into the Browser by clicking the Select mouse button on the interface icon in the Project Window or, if the interface is open, using the Select mouse button to click on the interface or any object in the interface.

2. Select File⇒Load from the Browser, or click on the Load icon in the Browser's icon bar.

The Load option in the Browser's File menu is sensitive only when an interface, an object in an interface, or an interface icon in the Interfaces Area of the Project Window is selected.

Only one interface can be loaded into the Browser at a time.

Selecting Objects in the Browser

There are several ways to select the objects displayed in the Browser.

If the interface loaded in the Browser is open, select or unselect operations in the Browser are duplicated in the interface. Any object you select or unselect in the Browser is simultaneously selected or unselected in the interface and vice versa.

To Do This	Do This
Select an object.	Click the Select button on the object.
Select an additional object.	Press and hold the Control key and click on more objects.
Select a group of objects.	Press the Select button and drag the mouse pointer over the interface icons.
Select all objects in the interface	Choose Edit⇒Select All from the Browser or choose Selected Objects⇒Select All in the Browser.
Cancel a selection of a group of objects.	Press the Escape key.
Unselect an object.	Hold the Control key and click on the object again.
Unselect all objects.	Click in an empty portion of the Interfaces area. You can also choose Edit⇒Deselect All from the Browser, or choose Selected Objects⇒Deselect All.

Note: Selecting an object also unselects all previously selected objects, unless you are holding down the Control key.

Duplicating Objects

To Duplicate an Object Using the Browser

1. Open the Browser and load the interface.
2. Select the objects to be duplicated in the Browser.
3. Click on the Duplicate icon in the Browser's icon bar, or choose Selected Objects⇒Duplicate, or choose Edit⇒Duplicate from either the Browser or the Project Window.

In the Browser, the duplicate is placed after the last object in the hierarchy. In the interface, the duplicate is offset to the lower right of the original.

When a parent is duplicated, it and all its children are duplicated. When the top-level interface in the Browser is duplicated, a new top-level interface is created.

Deleting Objects

To Delete Objects Using the Browser

1. Open the Browser and load the interface.
2. Select the object or objects to be deleted in the Browser.
3. Choose Selected Objects⇒Delete, or choose Edit⇒Delete from the Browser.

   A dialog box prompts you to confirm deletion.
4. Click OK or, with the mouse pointer in the Dialog box, press Return. To cancel deletion, click Cancel.

   The deleted object disappears from the interface and the Browser. The deleted object's children are also deleted from both windows.

Reparenting Objects

To Reparent Objects Using the Browser

1. Open the Browser and load the interface.
2. Select the objects to be reparented.
3. Press and hold the Adjust mouse button over the objects to be reparented.
4. Drag the Browser objects to their new location in the hierarchy. The pointer must be positioned directly over the intended parent.
5. Release the mouse button. The objects are reparented both in the interface and in the Browser.

If more than one Browser is open, you can reparent objects in one to parents in another.

Changing Your View of the Loaded Interface

You can view an interface loaded into the Browser in two formats: outline and tree. In the outline format, objects are arranged vertically, with child objects listed under their parent and indented to the right. In tree format, objects are arranged horizontally with parent and child objects connected by a thin line. Tree format is the default. Only one format can be active at a time. The active view format is unselected in the View menu. The icon for the active view format is also indented in the Browser's icon bar.

To Change the Browser's View

Click on the icon for the desired view in the Browser's icon bar, or choose it from the Browser's View menu. The view changes to the selected format. By default, every child object in a hierarchy is displayed.

To Hide Children

1. Select any parent object in the hierarchy.
2. Select View⇒Contract Node from the Browser, or double-click on the parent object, or choose Selected Objects⇒Contract Node in the Browser.

   The parent object's outline is thickened in the Browser to indicate that it has hidden children. If the parent object's name is showing, an asterisk is added.

To Show Children

1. Select any parent in the hierarchy with an asterisk indicator in it.
2. Choose View⇒Expand Node from the Browser, or double-click on the parent, or choose Selected Objects⇒Expand Node in the Browser.

3 Building Palettes

Overview

A palette is a storage area for reusable interface building blocks for your applications. Palettes can be shared across applications and between users. Palettes are composed of one or more categories, each category capable of storing any number of objects. The Ux palette, for example, contains categories for primitive objects, manager objects, menus, dialog objects, shell objects, and gadgets, plus the compound objects used in novice mode. Each of these categories holds the icons that represent the objects available in that category. Objects are identified with an icon, a label, or both.

Once stored in the palette, each object can be selected and easily added to the interface that you are creating or used to create a new interface. UIM/X is shipped with the Ux palette, containing all the Motif interface building blocks. This palette is loaded by default at start-up. For more information on the Ux palette, see the *UIM/X Motif Developer's Guide*.

About Palette Modes

A palette has two modes: Create and Edit. When a palette is in Create mode, selecting an object in the palette will allow you to make a copy of the palette item for use in your application. Edit mode makes available a series of commands that are used to manage the palette contents (adding and deleting objects, changing attributes, and so on). The Ux palette may also have objects or categories added, removed, or modified, in the same fashion as palettes that you create.

Selecting Create Mode

To select Create mode, choose Mode⇒Create from the palette. When you first create a new palette the mode will be set to Edit to facilitate initial setup. Once the setup is complete, the palette must be set to Create mode so that the objects in it may be used to create your application.

When you are in Create mode, the Edit menu and the Selected Objects pop-up menus are unavailable. All other menus are available.

Selecting Edit Mode

To select Edit mode, choose Mode⇒Edit from the palette. When you are in Edit mode, you can manipulate the categories in the palette and the objects in the categories. You cannot copy objects from the palette to your interfaces.

Creating an Object Library

The creation of a library of objects for use in your applications begins with the creation of a palette.

Creating a Palette

To create your own palette, select Create⇒Palette from the Project Window.

UIM/X creates and names a palette. The name it assigns is `palette`, with a number indicating the order of creation: `palette1` for the first palette you create. The filename of the new palette will be `palette1.pal`. This name can be changed by choosing File⇒Save As from the new palette.

All objects within a palette are stored in categories. By default, UIM/X creates an empty category when a new palette is created. You can easily add new categories to help organize your palette objects.

The default category is named `Untitled`. The category name can be changed by selecting Edit⇒Change Attributes from the new palette.

Figure 3-1 shows a new palette (with the default name `palette1`) and a category named `myCategory`.

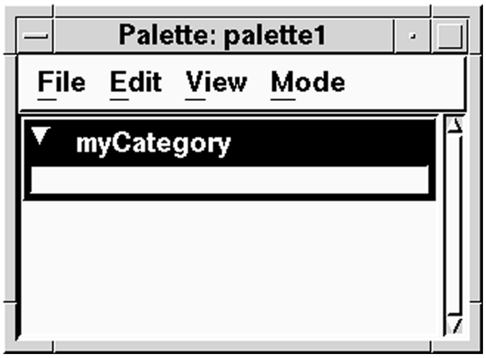


Figure 3-1 Palette with a Named Category

Storing Objects in a Category

Objects can be stored within the categories of a palette. You can drag an object to a palette from an interface or from the Browser. You can always drag objects into the palette—the palette can be in either Create or Edit mode.

To Drag an Object to a Palette

1. Select the object. You can select multiple objects to put into the palette. If the objects you select have children, the entire hierarchy must be selected.

   For example, if you have a Form with two Push Buttons on it, selecting the Push Buttons will make two entries in the palette. Selecting the Form and the two Push Buttons will make one entry (containing all three items) in the palette. Selecting the Form and only one of the Push Buttons is an error. UIM/X does not allow you to put a portion of an object hierarchy in the palette. Selecting the Form without selecting either of the two Push Buttons results in the same behavior as selecting the Form and both Push Buttons.
2. Drag the objects to the palette and drop them in a palette category.

Note: UIM/X never allows you to drop a fragmented hierarchy, such as a parent and only some of its children, or only a subcomponent of a menu to the palette. UIM/X displays an error dialog in either case.

Figure 3-2 shows the result of storing an object called `workingDialog` in the `myCategory` category.

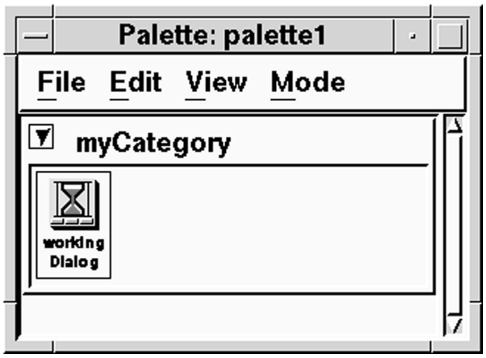


Figure 3-2 Palette with a Stored Object

Working with Palettes

When a palette is created it becomes part of the current project. Palettes can be manipulated in a variety of ways within the project. You can:

- Save the palette.
- Open and close palettes.
- Add an existing palette to a project.
- Insert one palette into another.

System palettes, which are loaded at start up, are never saved with the project. (System palettes are specified by setting the resource `UxStartingPalettes` in your Application Defaults.) They do not become part of the project, even when a system palette is saved under a new name with the Save As selection in the palette File menu.

However, you may wish to customize the objects offered by a system palette and make those customized versions available during project design. To do so, save the system palette under a new filename and specify that name when you set the `UxStartingPalettes` resource. The file will need to be saved in the directory specified by the `UxPalettePath` resource. See *"Loading System Palettes"* on page 11 for further information.

Alternatively, if the system palette file has write access, save the modified palette under its current filename, overwriting the original version.

Saving an Individual Palette

The palettes you open and create during a design session are saved with your project. You can also save palettes individually.

To Save a Palette

1. Choose File⇒Save As from the palette.

   A File Selection box prompts you to enter a palette file name.

   By default, the palette file is written to the current directory under the palette's current name. You can choose a different directory and enter a different name.
2. Click OK to save the file. Click Cancel to cancel the operation.

Alternatively, you can choose File⇒Save from the palette. If the palette has not already been saved, it functions exactly like choosing File⇒Save As. Once the palette has been given a name, choosing File⇒Save stores the palette using the current settings, without displaying a prompt.

File⇒Save As and File⇒Save are available in both Create and Edit Mode.

Note that when the Palettes Area is displayed in the Project Window, both File⇒Save and File⇒Save As are available in the Selected Palettes popup menu.

When you save a palette, the following information is saved in the palette file:

- Palette geometry (x, y, width, and height).
- Palette View setting.
- Palette Mode setting.
- All categories and objects.

Note: If you change only the size and position of a palette, you must choose File⇒Save As to save the new geometry settings in the palette file. Choosing File⇒Save does not modify the palette file if the geometry settings are the only changes made to the palette.

Displaying the Palettes Area

When you work with your own palettes, it is convenient to display the Palettes Area of the Project Window. This allows you to select different palettes and perform operations on them.

1. Choose View⇒User Palettes Area from the Project Window.

Closing a Palette

1. Choose File⇒Close from the palette.

   OR
2. Select the palette icon in the Palettes Area.
3. Choose Selected Palettes⇒Hide.

All palettes created during a session are saved when the project is saved, regardless of whether the palette is open or closed.

Showing a Palette

1. Point to the palette icon in the Palettes Area of the Project Window.
2. Double-click the Select mouse button.

   OR
3. Select the palette icon in the Palettes Area of the Project Window.
4. Choose Selected Palettes⇒Show.

Adding an Existing Palette to a Project

Palettes already saved with a project are automatically included when the project is opened. You can also add new palettes to your project.

To Add a Palette to a Project

1. Choose File⇒Open from the Project Window.
2. In the File Selection box that appears, specify the directory containing the palette and the name of the palette file.
3. Click OK.

   The palette interface will appear. If the Palettes Area is visible an icon will be added for the loaded palette.

Inserting One Palette Into Another

You can insert the contents of one palette into another.

1. Open the destination palette.
2. Select File⇒Insert from the destination palette.
3. In the File Selection box that appears, specify the directory containing the palette and the name of the palette file.
4. Click OK.

   All of the source palette’s categories and objects are written to the destination palette.

Deleting a Palette from a Project

You can remove a palette from a project.

1. Select the palette’s icon in the Palettes area of the Project Window.
2. Choose Selected Palettes⇒Delete.

   OR

2. Choose Edit⇒Delete from the Project Window.

Deleting a palette from a project does not delete the palette file itself.

Working with Categories in a Palette

The categories within a palette can be manipulated when the palette is in Edit mode.

Creating a Palette Category

You can create additional categories for storing objects.

1. Make sure the palette is in Edit Mode.
2. Select Edit⇒Create Category from the palette.

   UIM/X displays a dialog titled New Category where you name the category. UIM/X provides a default name `Untitled` suffixed with a number, to ensure that the name is unique within the current palette.
3. Enter a new name for the category, remembering that all category names for this palette must be unique.

   If a duplicate name was entered, a dialog will be displayed and the operation is cancelled.
4. Click OK to complete the operation. UIM/X creates a category inside the palette. The name you assigned is displayed in the category’s border. An arrow on the border opens and closes the category.

Selecting a Category

1. Make sure the palette is in Edit Mode.
2. Click on the category’s border with the Select mouse button.

   The category is highlighted.

To select more than one category, press the Control key while clicking on the categories. Alternatively, press and hold the Select mouse button and drag the pointer over the desired categories.

Once a category has been selected, you can change its attributes, delete it, or paste into it objects that are currently in the clipboard.

Changing a Category's Name

1. Make sure the palette is in Edit Mode.
2. Select a category.
3. Choose Edit⇒Change Attributes from the palette.
4. In the dialog box that appears enter a new name for the category.
5. Click OK to change the name.

Deleting Categories

1. Make sure the palette is in Edit Mode.
2. Select the category or categories to be deleted.
3. Choose Edit⇒Delete from the palette or choose Selected Objects⇒Delete.

   A dialog prompts you to confirm the deletion.
4. Click OK.

Opening and Closing a Category

Whether the palette is in Create or Edit mode, all categories can be opened and closed as required.

The expand arrow to the left of the category's name is a toggle button that expands and collapses the category. When a category is expanded, all the object icons it contains are visible. When a category is collapsed, only the category border with the category name is visible.

Working with Objects

The objects in a palette can be manipulated only when the palette is in Edit mode.

Changing Object Attributes

1. Make sure the palette is in Edit Mode.
2. Select the object. When changing attributes, you can select only one object at a time.
3. Choose Edit⇒Change Attributes from the palette.

The Attributes window appears, allowing you to change the object’s name, its icon, or both.

4. To change the object’s name, click on the Icon Name text field and enter a new name.

   To change the icon, click on the Icon Pixmap field and enter the file name of the desired icon. UIM/X recognizes pixmap or bitmap files.
5. Click OK.

Deleting an Object

1. Make sure the palette is in Edit Mode.
2. Select the object or objects to be deleted.
3. Select Edit⇒Delete from the palette or choose Selected Objects⇒Delete.

   A dialog prompts you to confirm the deletion. The delete operation cannot be undone.
4. Click OK.

   Deletion does not place the object icon in the clipboard.

Pasting an Object into a Category

You can use cut and paste to rearrange the objects in a category, or to move objects from one category to another.

1. Open the destination palette and put it in Edit Mode.
2. Use the Cut or Copy command to paste an object to the clipboard.
3. Select an object from the destination category or the destination category itself.
4. In the destination palette, choose Edit⇒Paste Before or Edit⇒Paste After, or choose Selected Objects⇒Paste Before or Selected Objects⇒Paste After.

   Paste Before will place the clipboard contents before the selected object or at the beginning of the selected category.

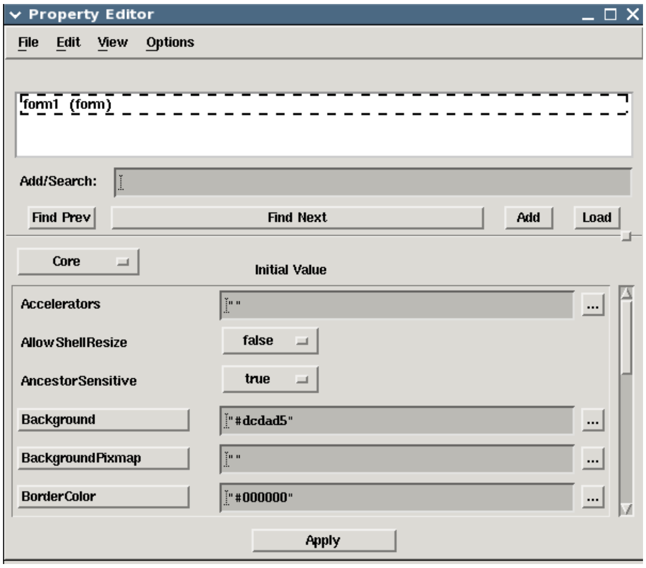
   Paste After will place the clipboard contents after the selected object or at the end of the selected category.

Setting Properties

4

Overview

The Property Editor displays the properties of an object and their initial values, and allows you to change those initial values. Properties can be changed for a single object or common properties can be changed for more than one object at a time.



form1 (form)

	Initial Value
Accelerators	" "
AllowShellResize	false
AncestorSensitive	true
Background	"#dcdad5"
BackgroundPixmap	" "
BorderColor	"#000000"

Figure 4-1 Property Editor

Opening the Property Editor

You can open the Property Editor empty or with an object loaded. To open an empty Property Editor, do the following:

1. Make sure no object is selected.
2. Choose Tools⇒Property Editor from the Project Window.

   The Property Editor appears, ready for you to load an object into it. If the Property Editor is open but hidden by other windows, this brings it to the top of your open windows.

To open the Property Editor with an object loaded, do the following:

1. Double-click on any object in your interface.

   OR
2. Click on any object to select it.
3. Choose Selected Objects⇒Tools⇒Property Editor.

   OR
4. Choose Tools⇒Property Editor from the Project Window.

   The Property Editor appears, loaded with your selected object.

Setting the Loading Options

After you open the Property Editor, you can choose from three possible loading options. These loading options let you customize how you load objects into an open Property Editor. The following table lists the available options:

Option	**Description**
Replace List	Loading objects into the Property Editor replaces any objects already present.
Add To List	Loading objects adds to the list of objects already in the Property Editor.
Automatic Load	Selecting objects automatically loads them into the Property Editor, replacing objects already present.

The Replace List and Add To List loading options support the standard ways of loading an object:

- Dropping the object in the Object List area.
- Entering the name of the object in the Add Object text field.
- Selecting the object (either in the interface or in the Browser) and then selecting File⇒Load from the Property Editor menu.
- Selecting the object (either in the interface or in the Browser) and then clicking on the Load icon in the Property Editor's icon bar.

The Automatic Load option provides one way to load objects: by selecting them, either in the Browser or in the actual interface.

Using Replace List

With the Replace List option, loading objects into the Property Editor replaces any objects already in the Property Editor. The Replace List option is the default loading option. Typically, you use the Replace List option when you want to edit objects one at a time.

To choose the Replace List option, click on the Replace List icon in the Property Editor's icon bar, or choose Options⇒Replace List from the Property Editor.

Using Add To List

With the Add to List option, each time you load an object it is added to the Object List. To remove objects from the Property Editor, select the objects you want to remove, then choose Edit⇒Remove Objects.

To choose the Add To List option, click on the Add To List icon in the Property Editor's icon bar, or choose Options⇒Add To List from the Property Editor.

Using Automatic Load

The Automatic Load option allows you to load objects by selecting them, either in an interface or in the Browser. To load multiple objects, you select multiple objects. To remove an object, you unselect the object.

When you select Automatic Load, the only way to load objects is by selecting them. Note that when Automatic Load is active, selecting Edit⇒Remove Objects cancels the Automatic Load option, and replaces it with the Replace List option.

To choose the Automatic Load option, click on the Automatic Load icon in the Property Editor's icon bar, or choose Options⇒Automatic Load from the Property Editor.

Removing Objects from the Property Editor

You can remove objects from the Object List in the following ways:

- In the Object List, select the objects to be removed, and then choose Edit⇒Remove Objects.
- Choose File⇒Reset to remove all objects currently displayed on the Object List. If you are using Automatic Load, File⇒Reset will cancel it with a warning message. The Object List changes to Replace List mode.

Searching for Widgets

You can search for existing widgets in your interface by typing part of the widget's name or placing pattern in the Add/Search field:

- Type the widget's name or pattern in the Add/Search field. For example, the following pattern "^form\d*" will allow you to find all widgets whose name starts with the text "form".
- Press Find Next. The Add/Search field will change to the matched widget.
- If you want to search further, press Find Next.
- To add/load a matched widget to the Resource Editor, press Add/Load.

Selecting the Property Category

To make it easier to find and edit properties, the Property Editor divides an object’s properties into categories. The Category Option menu lists the available categories, as shown in the following table.

Property Category	**Description**
Core	Properties such as X, Y, Width, and Height that are common to all objects.
Specific	Specific properties are those specific to that object class, such as the arrow direction for the Arrow Button.
Constraint	Constraint properties are those added to an object by its parent, if the parent provides functions for maintaining its children in a particular spatial arrangement. For example, children of a Form object receive additional properties that determine their position with respect to the Form and to each other. Constraint properties exist due to the parent object. If the child of a Form is moved to become a child of a Bulletin Board, its constraint properties disappear. Not all parents confer Constraint properties to their children.
All Resources	Lets you view and edit all of an object’s core, specific, and constraint properties at the same time.
Behavior	Behavior properties determine how the object reacts to callback events. For example, a Push Button has an `ActivateCallback` property, where you can specify the callback code that is executed when the button is pushed. Each callback has a corresponding `ClientData` property. The `ClientData` property is a pointer whose value is passed to the corresponding callback each time the callback is invoked, for example, each time you click on a Push Button.
Compound	Compound properties are properties defined by UIM/X. These properties allow you to control the editing operations that can be applied to an object.

Property Category	Description
Declaration	Declaration properties are specific to an object, such as its name, class, parent, and scope. Objects whose scope is defined as static will be declared statically in the generated code, while global objects will be declared non-statically.
All	Lets you view and edit properties of all categories at the same time.

Setting Properties

A number of guidelines will help you to understand how to set properties. The first is that you specify the value in the same way you would in an X resource file. For example, to set a color, rather than allocating Color Map entries and setting pixels you can use `"SlateBlue"` or `"#6a6a5a5acdcd"`. UIM/X also has a Color Viewer that allows you to specify colors. The Color Editor can be opened from the Property Editor. To specify a list of strings, place items in a string separated by commas.

The second guideline gives UIM/X tremendous power: property values are all code expressions. You can put into a property any expression that returns the correct type, including constants, global variables, and functions that take arguments. You can use this technique to set values dynamically at run time.

There are a number of standard property value types:

Property Value	Description
Integer	Either an integer or an expression that evaluates to an integer.
Float	An integer, a float, or an expression that evaluates to an integer or a float.
String	A string constant enclosed in quotation marks (" "), or an expression that returns a `char*`.
Boolean	Either the character string `"true"` or `"false"`, or an expression that returns a `char*`.

Property Value	**Description**
Color	Any color name recognized by your server and enclosed in quotes; an expression that returns a `char*`; or a value in the rgb format where hexadecimal digits represent the r, g, and b values, for example: `“#6a6a4c4c8d8d"` (MediumSlateBlue). For properties that require colors as their values, the property name appears as a button on the Property Editor. Choosing this button opens the Color Viewer.
FontList	Any font list with font names recognized by your X server. For properties that require font names as their values, the property name appears as a button on the Property Editor. Choosing this button opens the Font Viewer.
Pixmap	Any pixmap found by your application. For properties that require pixmap names as their values, the property name appears as a button on the Property Editor. Choosing this button opens the Icon Viewer.
Enumerated Type	Many object properties have a small number of strings as possible values, listed in an option menu beside the property name. For example, the value for the EditType property can be the string constant `"edit"`, `"append"`, or `"read"`.
Code	Any legal expression or statement of the correct type for the property.

Setting Pixmap Properties

Interfaces you create with UIM/X often use pixmap files. For example, a Push Button might feature a pixmap indicating its purpose, rather than a text label. Rather than storing the pixmap itself in the object, UIM/X stores the name of the file containing the pixmap. At run time, UIM/X and applications with interfaces generated by UIM/X look for the file in specific directories.

These directories are listed in the path list `UxBitmapPath` (a *path list* is a data structure that lists a set of search paths). `UxBitmapPath` lists the default search path for pixmap files:

1. `*uimx_directory*/icons`
2. The current directory
3. The developer's `$HOME` directory
4. `/usr/include/X11/bitmaps/`*app_class_name*, where *app_class_name* is the class name of the application (the second argument passed to `XtAppInitialize()` in the generated main program file).

   For UIM/X, *app_class_name* is `Uimx3_0`. For applications generated by UIM/X, *app_class_name* is the value entered in the Application Class name field of the Program Layout Editor.
5. `/usr/include/X11/bitmaps`

Note: On Solaris, the `include/X11/bitmaps` directory is found within `/usr/openwin`, rather than within `/usr`.

You can modify `UxBitmapPath` using the Ux Convenience Library functions `UxInitPath()` and `UxAddPath()`. A related function, `UxExpandBitmapFilename()`, takes a pixmap file name and looks for that file in the search path defined by `UxBitmapPath`. It returns the file name prefixed by its directory path.

Default Property Values

When you open the Property Editor for an object to find out what its default values are, UIM/X asks the object for those values. There are a number of ways an object property gets a default value—each has certain implications.

First, if properties are not specified in any other manner, an object class has default values for all properties.

Second, you may have specified a property value by setting an X resource. Note that values set in this way are transferred to the end user of the generated application only if that user has a corresponding setting of the resource.

Third, the object may determine the value dynamically depending on the values of other properties. For example, when you change the background color of a Push Button, the top and bottom shadow colors—if not explicitly set—dynamically change to colors that match well. Another special case to be aware of is the constraint attachment properties—left, right, top, and bottom—for children of Forms. The default values for these are respectively: `attach_form`, `attach_none`, `attach_form`, and `attach_none`.

When you change the right attachment to `attach_form`, you might expect the others to stay the same and for the right edge of the object to move to the Form. Instead, new default values are calculated for the other three attachments. The new values are: `attach_none`, `attach_form`, `attach_form`, and `attach_none`.

Using Resource Files to Set Property Values

Often it is desirable to have the properties for all or a portion of the objects in an interface be the same value. For example, you may want the background color of every object in an interface to be red. One way to accomplish this is to load all of the objects in the interface into the Property Editor and change the value of the property. As an alternative, you may use the standard mechanism in the X Window system. In the appropriate user resource file, you may set values as shown in the following table:

Resource	**Description**
appName.objectName.propertyName: value	Sets the property *propertyName* of the object *objectName* to *value***.**
*appName.objectName*propertyName: value*	Sets the property *propertyName* of the object *objectName* and all its children to *value***.**
*appName*propertyName: value*	Sets the property *propertyName* of all objects to *value*.

Resource	**Description**
appName.objectClass.propertyName: value	Sets the property *propertyName* of all objects of class *objectClass* to *value***.**
appName is the name of the application or class.	

Note: The above example sets default values and *will not* apply to values set explicitly in the application.

Setting values with too wide a scope (for example, `*LeftAttachment`) may adversely affect interfaces.

Changing a Property Source

Each property has a source, indicating how the property is treated when you generate code. Its meaning depends on the category of the property as shown in the table below.

By default, the source column is hidden. To see the Source menu, unselect the Hide Source toggle button in the Property Editor View menu.

The Property Source menu lets you set the following:

Properties	**Source**	**Description**
Core Specific Constraint	Default	Determined by object default. No value appears in the generated code.
	Public	Set in a generated resource file.
	Private	Set in the object creation section of the generated code.
	Lock	Cannot be edited further without first unlocking it.

Properties	Source	Description
Behavior	None	No callback.
	Extern	Callback is specified as the name of a function. The function is defined elsewhere in the auxiliary declarations or in a separate source file.
	Static/Virtual	Callback specified is actual code to be executed.
	Lock	Cannot be edited.
Name	Global	Swidget variable declared global in generated code.
	Static	Swidget variable declared static in generated code.
	Lock	Cannot be edited.
Parent	Private	Parent determined by code.
	Lock	Cannot be edited.

Notes on Property Sources

- Some properties cannot be set in a resource file, in which case the `Public` entry in the source menu is insensitive.
- When a property is locked, a lock symbol appears beside the value field, the Initial Value text fields and option menus are insensitive, and an `Unlock` entry appears in the Source option menu.
- Some properties are locked and the lock symbol is grayed-out. This means the property is set via an editor external to the Property Editor (such as the Menu Editor), or that the properties are set dynamically by a Manager object. In both cases you cannot unlock the property.
- When you change a resource value with a Default source, UIM/X automatically changes the source to `Private`.
- When you change a callback, UIM/X changes the source to `Static` (or `Virtual` in C++) or `Extern`, depending on whether the callback function was declared `static` or `extern` in the Callback Editor.

Editing Properties for Multiple Objects

The Property Editor also allows you to edit properties for more than one object at a time. If more than one object appears in the Object List, only the properties those objects share appear in the Properties area. Changes made to any shared property apply to all objects on the Object List. For example, to change the highlight color of all objects in an interface, load all the objects into the Property Editor, change and then apply the `HighLightColor` property value.

Note: Shared properties are those that have the same point of definition in the Motif widget class hierarchy. For example, the `LabelString` property of a Push Button object is defined by the Motif Label class. The `LabelString` property of a Push Button gadget is defined by the Motif LabelGadget class. Although the name of the property is the same, the Push Button and the Push Button gadget do not actually share the `LabelString` property. The `LabelString` property is not shown in the Property Editor when a Label and a Label Gadget are placed together in the Object List.

A property shared by several objects but with values that differ from object to object is highlighted by a not-equals sign to the left of the Initial Value field. For example, consider two objects that share the `Background` property. For one, `Background` is set to Blue; for the other, Grey. If both objects are loaded into the Property Editor, a not-equals sign appears to the immediate left of the `Background` property and the Initial Value field is blank.

If the sources of a shared property differ, the not-equals sign appears in the Source column. For example, the shared property may be set to `Public` in one object and to `Private` in another. When the values differ, the Initial Value field will be blank and the not-equals sign will appear in the Source column.

Changing Object Names

The name of an object is one of its Declaration properties. The Name property is not a string, but the name of the variable that holds a pointer to the object. In UIM/X and in generated code that uses the Ux Convenience Library, this variable holds a swidget pointer. In generated Xt code, the variable holds a widget pointer. When Ux

Convenience Library C++ Bindings are used, the variable holds an object of the appropriate Motif wrapper class (which contains a swidget pointer).

(A swidget is a shadow widget, a data structure used by UIM/X to represent the actual object.)

Changing the Scope of an Object

The scope of an object can be set to `Static` or `Global` depending on how you want the object to be accessed. This affects how the object variable is declared in the generated code. If it is static, then the variable is declared statically; if it is global, the variable is not declared statically and can be referenced from other files by means of an `extern` declaration.

To Change the Scope of an Object's Name from Static to Global

1. In the Property Editor, set the Source to `Global` for the `Name` property (in the Declarations properties).

Note: If the object's name is set to `Global`, you cannot use multiple copies of the interface.

Reparenting Objects

An object can be reparented by loading the object into the Property Editor, choosing the Declaration properties from the Category option menu, and then entering the name of the new parent object in the Parent field.

Explicit and Implicit Shells

By default, all top-level interfaces, whether they have explicit or implicit shells, are created with the value `UxParent` in the Parent field. `UxParent` is an argument passed to the interface function of the interface and is the parent of that interface.

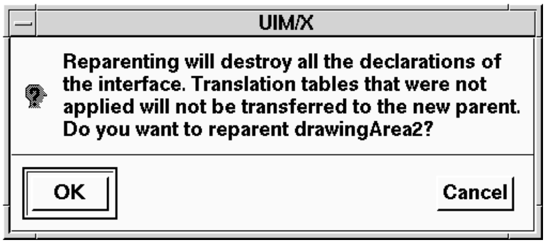
In a number of special cases, top-level interfaces do need a real parent. For example, you may wish to make all top-level shells the children of a single application shell so that they all iconify with the application shell. Another example is making a dialog shell the child of the interface from which it is called so that input can be locked out from that interface (setting the `DialogStyle` to `dialog_application_modal`).

Reparenting Explicit Shells

When you change the parent of an explicit shell, initially you will see no effect. It is only when you create and pop-up the interface using the interface function that the change will take effect.

Reparenting Objects with Implicit Shells

When you attempt to reparent an object with an implicit shell statically, a warning dialog displays the following message:


UIM/X

Reparenting will destroy all the declarations of the interface. Translation tables that were not applied will not be transferred to the new parent. Do you want to reparent drawingArea2?

Figure 4-2 Warning Dialog

If you click OK, the manager will become a child of the parent you specified, as expected. This is often used when you begin polishing the interface or window manager interaction, changing the properties of the explicit shell. Typically you reparent an implicit shell manager to an explicit shell.

Dynamic Parenting

Dynamic parenting is made possible when a valid code expression is entered into the Parent field of the Property Editor. When you edit the interface to which you have dynamically added the implicit shell manager, you will see a message that informs you that you will delete and/or recreate all dynamically created objects in the interface. The message will ask if you want to continue. By choosing OK, the implicit manager will be recreated as a top-level.

Reparenting Objects

Child objects can also be reparented. When created, their parent is the object under the mouse pointer when the mouse button is released. However, design needs may dictate that the child of one object become the child of another object.

For example, consider a top-level Bulletin Board interface, named `bulletinBoard1`, with two Push Buttons as its children. You decide that the Push Buttons should be managed by a RowColumn object. The RowColumn object, named `rowColumn1`, is created in the same Bulletin Board interface. The two Push Buttons are loaded into the Property Editor. In the Parent field of the `Declaration` property for the Push Button objects, change the name in the Parent field from `bulletinBoard1` to `rowColumn1`.

The Push Button objects are transferred from the Bulletin Board object to the RowColumn object.

Child objects can also be reparented in the Browser and in the actual interface. Use the Adjust mouse button to select the object to be reparented and drag it until it is directly over its new parent. The child is reparented when the Adjust button is released.

To Reparent an Object in the Property Editor

1. In the Parent field, enter the name of the new parent.

The new parent can be any object that accepts children, whether in the current interface or in another interface. The new parent can be `NO_PARENT`, that is, the object can be promoted to top-level status.

To Reparent a Child Object in the Browser

1. Drag the object to its new parent and drop the object.

   The object will be reparented to the new parent. Because only one interface at a time can be displayed in the Browser, the new parent must itself be a child in the same top-level interface.

Promoting a Child Object to Top-Level Status

Promoting a child object to top-level status is a form of reparenting. It is most commonly used when a child object is to serve as a Component.

Any child object, except a menu and a gadget, can be promoted to top-level. Promotion to top-level automatically adds the default implicit shell to the newly promoted object.

To Promote a Child Object to Top-Level Status

Set the `Parent` property to the value `NO_PARENT`, or drag the object onto the desktop.

In both operations, the child object is removed from the interface where it resided. The default implicit shell is applied automatically to the newly promoted object.

Note: Menus and gadgets cannot be promoted to top-level status.

Specifying Callbacks and Connections

5

Overview

Adding callbacks to an object is done in the Callback Editor or Connection Editor. The Callback Editor is available in the Behavior category of the Property Editor. Callbacks consist of C or C++ code.

You have a great deal of flexibility in the code you can use in a callback. You can use:

- The callback arguments (they are already declared for you).
- Ux Convenience Library function calls—the names of objects in the interface can be used directly with these calls (they are declared for you).
- Xm, Xt, or X functions (they are already linked in).
- Calls to your compiled functions and references to your compiled global variables if you have added them to UIM/X.
- Code that has been loaded into the Interpreter. Note that the Interpreter has three modes: K&R C, ANSI C, and C++ (the default).

Note: If you specify an external function for the source of the callback code, you must declare the function as extern in the Declaration Editor global variables section before applying the callback. Otherwise the external function will not be recognized.

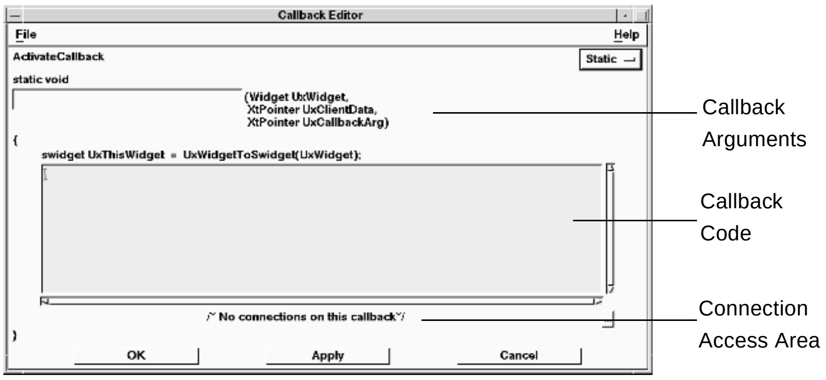
The Callback Editor

The Callback editor allows you to enter code to specify the behavior you want in your application.

Opening the Callback Editor

1. Select an object and load it into the Property Editor.
2. Select the Behavior category of Properties.
3. Click on the edit button (…) beside the any callback property.

The Callback Editor appears, as shown in Figure 5-1.



```
ActivateCallback
static void
                (Widget UxWidget,
                 XtPointer UxClientData,
                 XtPointer UxCallbackArg)
{
    swidget UxThisWidget = UxWidgetToSwidget(UxWidget);

            /* No connections on this callback */
}
```

Figure 5-1 Callback Editor

Opening the Connection Editor from the Callback Editor

The Connection Access area of the Callback Editor indicates whether any connections have been made from the callback you are editing. Clicking on the (...) button in the Connection Access area brings up the Connection Editor, allowing you to add, edit, or delete connections from that callback.

Using Callback Arguments and Variables

When specifying behavior in the Callback Editor, all standard Motif callback arguments are available for use. These are shown in Figure 5-1, and explained below:

Argument	Description
UxWidget	The widget argument to the callback.
UxCallbackArg	The callback argument to the callback as supplied by the object. This argument must be cast to the appropriate type.
UxClientData	The client data argument to the callback. If a value was specified for the client data of this callback in the object's Property Editor, this argument will have the specified value. Otherwise, it will be equal to `UxContext`.

In addition, two variables are available:

Variable	Description
UxThisWidget	The swidget corresponding to the object.
UxContext	`UxContext` is a pointer to an interface-dependent context structure.

Making Connections

The Connection Editor establishes a behavioral connection between a source and target object by automatically creating snippets of callback code based on user-specified criteria.

Each connection is defined as a rule of the form: For *Source Object*, on *Callback*, perform *Method* on *Target Object*.

An example of a simple behavioral connection might be:

> For `pushButton1` (source), on ActivateCallback (callback), use SetBackground (method) to change the color of `text1` (target).

Connections can only be established between objects of the same interface. That is, connections across interfaces are impossible.

Opening the Connection Editor

You can open the Connection Editor in three different states; with a source object loaded, with both source and target objects loaded, or with no object loaded. You open the Connection Editor as follows:

1. Choose Tools⇒Connection Editor from the Project Window.

   If no object was selected in your interface, an empty Connection Editor appears. If an object was selected, the Connection Editor appears with that object loaded as the source.

   OR

1. Drag the mouse pointer (using the Select mouse button) from a source object to a target object while depressing the Shift key.

   A line drawn between the source and target objects appears during this operation to indicate the intended connection.

2. Release the mouse button.

   The Connection Editor appears, loaded with your chosen source and target objects.

   OR

1. Choose Selected Objects⇒Tools⇒Connection Editor.

   The Connection Editor appears with the selected object loaded as the source.

   OR

1. Click on the Text Editor (...) button in any Callback Editor’s Connection Access area.

   The Connection Editor appears with the selected object loaded as the source.

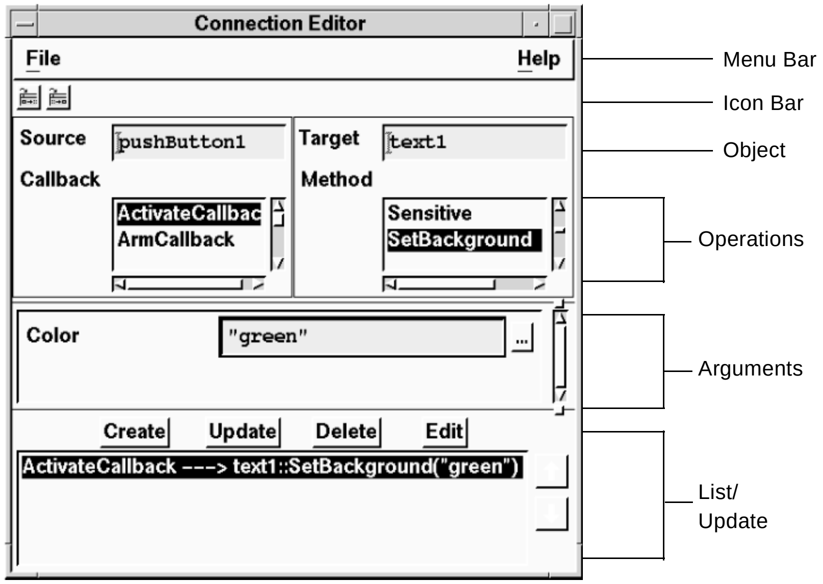
   OR

1. Choose Edit⇒Connection From or Edit⇒Connection To from the Menu Editor.

   The Connection Editor appears, loaded with your chosen menu object as the source or target, respectively.

About the Connection Editor

The Connection Editor comprises six main areas, as illustrated in Figure 5-2.



ActivateCallback ---> text1::SetBackground("green")

Figure 5-2 Main Areas of the Connection Editor

- *Menu Bar*: Provides a File menu to load source and target objects and reset or close the Connection Editor, and a Help menu to provide online help.
- *Icon Bar*: Provides icons for loading source and target objects.
- *Object area*: Enables selection of Source and Target objects.
- *Operations area*: Enables selection of the type of callback and method.
- *Arguments area:* Displays the values of the arguments for the currently selected method (including the return value, where applicable), and allows the user to change these values. Default values are provided initially.
- *List/Update area:* Provides a display area for the created connections and enables selection for subsequent actions using command buttons.

Loading a Source

To load a Source object:

Select an object in your interface, then choose File⇒Load Source or click on the Load Source icon in the Connection Editor.

OR

Drag an object from your interface or from the Browser and drop it in the Source field.

OR

Type the object name into the Source text field and press Enter.

The selected object and its associated callbacks are then loaded into the Connection Editor.

Loading a Target

To load a Target object:

Select an object in your interface, then choose File⇒Load Target or click on the Load Target icon in the Connection Editor.

OR

Drag an object from your interface or from the Browser and drop it in the Target field.

OR

Type the object name into the Target text field and press Enter.

The selected object and its applicable methods are then loaded into the Connection Editor.

Defining Connections

With both Source and Target loaded, you can begin to define a connection. To define a connection, you must select a Callback and a Method from the Operations area:

1. Click on a callback in the Callback scrolled window.
2. Click on a method name in the Method scrolled window.

When applicable to the selected method, the default values of the method's associated arguments (including a return value, where applicable) are displayed in their respective text fields in the Arguments area. You can change the arguments by typing in new values or by selecting a (...) button to invoke the Text Editor.

3. Click on the Create command button.

   The created connection is displayed and highlighted in the list section of the Connection Editor's List/Update area, for example:

```
ActivateCallback--->text1::SetBackground("green")
```

To create additional connections, repeat the above steps. Always click on the Create command button to create a new connection. The Connection Editor will add the new connection to the end of the list rather than replace the existing (highlighted) connection.

To duplicate a connection, highlight the connection and select Create. The connection is duplicated and appended to the list.

Modifying Connections

With the connection highlighted, you can perform further operations by selecting a command button in the List / Update area or selecting a different type of callback or method from the Operations area.

To Modify a Connection

1. If the connection is not already selected, select it from the list by clicking it, then click Edit.
2. Select a callback and/or method from the Operations area, as earlier described. Edit the argument values if you like.
3. Click on the Update command button.

The newly created connection appears in the list section, replacing the previously selected connection.

To Modify Arguments

1. Click on a connection to select it, then click Edit.

   The current values of the arguments appear in their respective text fields in the Arguments area.
2. Type in a new value for the argument or select the (...) button adjacent to the text field to invoke the Text Editor.
3. Click on the Update command button.

The connection is updated with the new argument value.

To Delete a Connection

1. Click on a connection to select it.
2. Click on the Delete command button.

The connection is erased from the list.

Types of Connections

When the target object is the instance of a component, the list of methods shown in the Connection Editor is the list of interface methods defined for that component. The code segment generated for a connection to an interface method consists of a call to that method.

When the target object is a Motif swidget, a special list of methods called *swidget methods* is shown. Swidget methods are not methods as such, but more like macros that expand into the appropriate code segment during code generation. The code segment generated for a swidget method is determined by the chosen code generation option. For example, the following connection:

```
ActivateCallback--->text1::SetText("Hello World!")
```

will generate one of the following code segments, depending on the type of code being generated:

Xt code	XtVaSetValues(text1, XmNvalue, "Hello World!", NULL);
Ux code	UxPutText(text1, "Hello World!");
C++ Bindings code	text1.SetText("Hello World!");

When the target object is the top-level swidget of an interface, both the swidget methods and the interface methods are shown. This mechanism allows any swidget within an interface to make a connection to the interface methods of its parent interface.

Some methods accept a return value. If an interface method accepts a return value, it is optional, but if a swidget method accepts a return value, one must be supplied. A return value supplied for a connection must be a variable of the proper type visible within the scope of the corresponding callback. This variable is assigned the result of the connection when it is triggered.

Closing the Connection Editor

To close the Connection Editor:

Choose File⇒Close from the Connection Editor.

OR

Double-click on the Connection Editor's Window menu button.

OR

Press Alt+F, then press C with the Connection Editor active,

OR

Press Alt+F4 with the Connection Editor active.

The Relationship Between Callbacks and Connections

When a connection is established between two objects, appropriate code is automatically added to the specified callback of the source object. This code comes after any code added to the callback by the Callback Editor, and is added in the order that the connections are displayed in the Connection Editor.

For example, consider an interface containing a top-level Form, a Push Button, a Text Field, and a Label, where the interface has a method called MyMeth, and the Push Button has the following code in its `ActivateCallback`:

```
char *text;
cout << "Transferring text." << endl;
```

If the following connections are made from the Push Button:

```
ActivateCallback--->textField1::GetText()
ActivateCallback--->label1::SetLabelString(text)
ArmCallback--->form1::MyMeth(42, &UxEnv)
```

where the first connection in the list specifies “text” as the return value, and C++ code is generated with Ux Convenience Library C++ Bindings enabled, the following two callbacks are generated for the Push Button:

```
void _UxCform1::activateCB_pushButton1(
     Widget wgt,
     XtPointer cd,
     XtPointer cb)
{
     Widget UxWidget = wgt;
     XtPointer UxClientData = cd;
     XtPointer UxCallbackArg = cb;
     swidget UxThisWidget;

     UxThisWidget = UxWidgetToSwidget( UxWidget );
     {
     char *text;
     cout << “Transferring text.” << endl;

     // Connection code
     text = textField1.GetText();
     label1.SetLabelString((text));
     }
}
```

```
void _UxCform1::armCB_pushButton1(
     Widget wgt,
     XtPointer cd,
     XtPointer cb)
{
     Widget UxWidget = wgt;
     XtPointer UxClientData = cd;
     XtPointer UxCallbackArg = cb;
     swidget UxThisWidget;

     UxThisWidget = UxWidgetToSwidget( UxWidget );

     // Connection code
     MyMeth(42,&UxEnv);
}
```

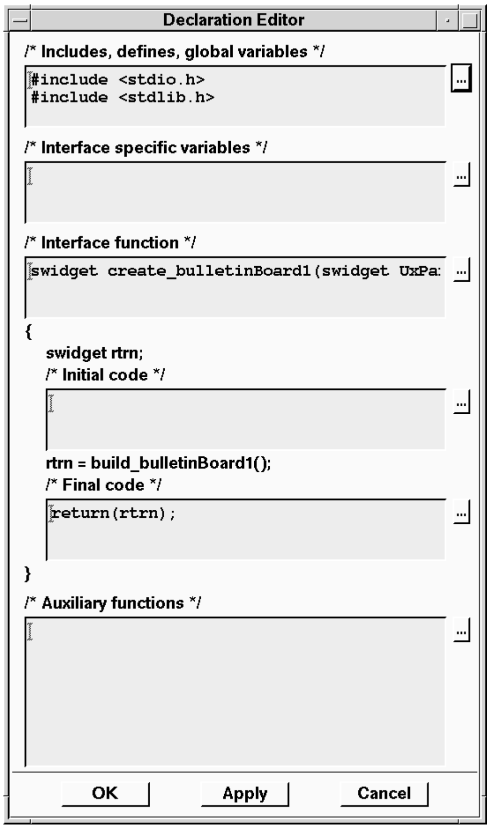
Editing Interface Code 6

Overview

UIM/X generates the code for each interface into a separate source file. The Declaration Editor acts as a template for the generated code.

The Declaration Editor allows you to edit portions of the file which will be generated.

Notice the standard function call declaration and the curly braces shown on the Declaration Editor which mark the beginning and the end of the create function. All text that you enter in the text fields will appear directly in the generated file, with minor exceptions (such as prefixing parameter and interface-specific variable names with `_Ux`).



```
/* Includes, defines, global variables */
#include <stdio.h>
#include <stdlib.h>
/* Interface specific variables */
/* Interface function */
swidget create_bulletinBoard1(swidget UxPa:
{
    swidget rtrn;
    /* Initial code */
    rtrn = build_bulletinBoard1();
    /* Final code */
    return(rtrn);
}
/* Auxiliary functions */
```

Figure 6-1 Declaration Editor

Opening the Declaration Editor

1. Select the interface’s icon in the Project Window.
2. Choose Tools⇒Declaration Editor from the Project Window, or choose Selected Interfaces⇒Tools⇒Declaration Editor.

   OR

1. Select the interface by clicking on it, or by clicking on an object within it.
2. Choose Tools⇒Declaration Editor from the Project Window, or choose Selected Objects⇒Tools⇒Declaration Editor.

Modifying Code with the Declaration Editor

Following the code guidelines established for UIM/X, enter text in the appropriate text fields for the sections of the code that you want to modify. The Text Editor (...) button accesses the Text Editor. The Text Editor offers more visible space for entering code than the Declaration Editor.

Once you have entered your code, click on OK or Apply in the Declaration Editor. This new code will be included when code for the interface is generated.

Reserved Words

You should not use the words `parent` or `name` anywhere in code specified in the Declaration Editor.

General Code Guidelines

When entering code into the Declaration Editor, keep in mind the following guidelines:

Interface Functions	You can call the Interface Function of another interface. It should be declared `extern` since it will appear in a different file.
Global Variables	Any global variables defined in the Declaration Editor of another interface can be used. They should be declared `extern` since they will appear in a different file.
Auxiliary Functions	You can call any non-static functions defined in the Auxiliary Functions section of the Declaration Editor of another interface. They should be declared `extern` since they will appear in a different file.

C++ Code Guidelines

When you wish to include C++ header files while in ANSI C mode or K&R C mode, the files must be properly protected as follows:

```
    #ifdef __cplusplus
    #include <Alok.h> // C++ header file
    #endif /* __cplusplus */
```

All code in the Declaration Editor will be written as is to the generated C++ file. Therefore, if you declare C variables or include C specific header files, you must use the C++ linkage specification statement as follows:

```
#ifdef __cplusplus
extern "C" {
#endif
/* C declarations and #includes entered here */
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

Declaring Global Variables

UIM/X writes the declarations of global and static variables at the beginning of the source file generated for the interface. You can reference global variables from other files if you have first declared the global variables to be `extern`.

The following variables are declared globally when the source file is generated:

- Variables specified as global in the Declaration Editor.
- Variable names for swidgets whose scope is set to global in the Property Editor.

Global variables and constants defined in the Includes, Defines, Global Variables section of the Declaration Editor may be used in code for objects of the corresponding interface, and in other fields on the Declaration Editor.

Note: Do not define a structure in the Globals section of the Declaration Editor having the same name as any of the interface specific variables. This could result in an unwanted macro expansion. See *“Restrictions Concerning Interface-Specific Variables”* on page 72 for more information.

If you want to use functions in code, the appropriate include files must be entered here. Apply the changes so that the Interpreter will recognize the functions.

The Includes, Defines, Global Variables section also allows you to:

- Specify include files using the `#include` directive.
- Define constants using the `#define` directive.
- Declare global variables.

Declaring Global Variables with C++ Bindings

When Ux Convenience Library C++ Bindings are used, swidgets are declared as objects of the appropriate Motif wrapper class. Therefore when making an `extern` declaration of a global swidget in another interface, the appropriate C++ wrapper class name must be specified. Because this type of declaration is incompatible with that needed when Ux Convenience Library C++ Bindings are not in use (in which case all objects are declared as type `swidget`), it is best to provide a conditional declaration. For example:

```
#if defined(__cplusplus) && !defined(XT_CODE) \
                          && !defined(UX_NOBINDINGS)
   extern UxPushButton pushButton1;
#else
   extern swidget pushButton1;
#endif
```

Declarations made in this way will work regardless of whether or not Ux Convenience Library C++ Bindings are used.

Declaring Interface-Specific Variables

In the generated code, interface-specific variables are declared as elements of an allocated structure (the context structure).

The following variables are declared as elements of the context structure:

- Variables specified as interface-specific in the Declaration Editor. Any variables entered in the Interface Specific section will be added to the context structure.
- Variable names for swidgets whose scope is local.
- Variable names for parameters passed to the Interface Function, as specified in the Declaration Editor.

Unlike the global variables, the interface-specific variables are specific to each interface. Interface-specific variables may therefore have a different value in each copy of the interface. This is implemented by means of the context structure.

Note: In generated C++ code, the context structure is the interface class itself. Therefore, interface-specific variables become member variables of the class.

Restrictions Concerning Interface-Specific Variables

If you encounter strange syntax errors when applying changes in the Declaration Editor, it is most likely that you are having problems resulting from a macro expansion. Note that the existence of the context structure macros in the generated code makes it impossible to declare a local variable in your callback code having the same name as one of the swidgets, instance-specific variables, or arguments passed to the Interface Function. If you attempt to do so, the macro expansion would result in a syntax error.

Similarly, do not define a structure in the Globals section of the Declaration Editor having the same name as any of the interface specific variables. This also could result in an unwanted macro expansion.

Auxiliary functions in C++ are generated as stand-alone functions, while interface-specific variables are generated as part of the interface class. The result is that auxiliary functions cannot use any of the interface-specific variables. See Appendix for more information.

Renaming the Interface Function

The Interface Function creates and potentially displays the interface. The default name for this function is obtained by prefixing `popup_` or `create_` to the name of the top-level object.

For example, if you create an interface whose top-level object is named `bulletinBoard1`, the generated file will contain a function `popup_bulletinBoard1()` or `create_bulletinBoard1()`. You can rename this function, using the Declaration Editor, to any legal function name.

Changing the Return Value of the Interface Function

The Interface Function, by default, returns the top-level object pointer (of type `swidget`). However, you can modify the function to return anything you want. This is done in the Declaration Editor.

For example, you might change the return type of the following Interface Function from:

```
    swidget create_bulletinBoard1(swidget UxParent)
```

to:

```
    Widget create_bulletinBoard1(swidget UxParent)
```

provided you also changed the return statement in the Final Code area from:

```
    return(rtrn);
```

to:

```
    return(UxGetWidget(rtrn));
```

Adding Parameters to the Interface Function

You can also specify parameters which are to be passed to the Interface Function by means of the Declaration Editor.

A typical example is to pass a character string which is to appear on the interface. For example, if the Interface Function is:

```
swidget create_bulletinBoard1(swidget UxParent)
```

You could change it to:

```
swidget create_bulletinBoard1(swidget UxParent,char *s)
```

Since this is a standard function declaration, all rules about declaring arguments apply.

Note that the name of the parameter must differ from the following names:

- Any objects in the same interface.
- Any variables declared in the Interface Specific section.

Otherwise, the generated code *will not* compile.

Adding Initial and Final Code to an Interface

In the Declaration Editor, you can specify initial and final code. Initial code is executed before the interface is created; final code after the interface is created.

Note: Initial and final code is only executed when an explicit call to the constructor of the interface is made. This means that it is not automatically executed when you click Apply in the Declaration Editor or when you switch to Test mode.

To test your initial and final code, you can switch to Run mode, or you can evaluate the Interface Function in the Interpreter window.

In initial code, you would initialize the values of interface-specific and global variables and make use of any parameters passed to the Interface Function.

A typical use of the Initial Code Field is to initialize interface-specific variables referenced when creating objects. For example, you might want to pass in the dimensions of the top-level object of the interface. These would be assigned to interface-specific variables in the Initial Code section. The Property Editor for the top-level object would reference the variables for the width and height properties.

When the resource `InterfaceFunctionType` is set to `create_` the default for the Final Code section is:

```
return(rtrn);
```

When the resource `InterfaceFunctionType` is set to popup_ the default for the Final Code section is:

```
VisualInterface_Manage(rtrn, &UxEnv);return(rtrn);
```

The `rtrn` value is the swidget of the top-level object of the interface.

Specifying Auxiliary Functions

Following the Interface Function are any auxiliary functions which you have specified in the Auxiliary Functions section of the Declaration Editor. This is a convenient spot to put any small routines that handle some aspect of the interface and are to appear in the same source file as the interface code. For example, when running an application as a subprocess, you may need a function to handle the data sent back from the application; this function can be defined as an Auxiliary Function in the Declaration Editor.

Note: Auxiliary functions in C++ are generated as stand-alone functions, while interface-specific variables are generated as part of the interface class. The result is that auxiliary functions cannot use any of the interface-specific variables. See *"Restrictions Concerning Interface-Specific Variables"* on page 72 for more information.

Note: Local variables defined in auxiliary functions must not have the same name as any parameters added to the Interface Function. This could result in an unwanted macro expansion in the auxiliary function. See *"Restrictions Concerning Interface-Specific Variables"* on page 72 for more information.

Linking Interfaces Together

Once you have created and added behavior to your interfaces, you will frequently want to link them together by having a callback in one interface create and display another interface.

When UIM/X generates code for an interface, it provides a function which is used to create the interface and, optionally, display it. This function is called the *Interface Function* and can be customized using the Declaration Editor.

UIM/X offers you a choice of two kinds of Interface Functions, a *create* function and a *popup* function. Both functions create the interface, but the popup function displays the interface as well. By default the name of the Interface Function is obtained by prepending `create_` or `popup_` to the name of the top-level object. This name can be modified using the Declaration Editor. For an interface whose top-level object is `form1`, the default names for the create and popup functions are `create_form1()` and `popup_form1()`.

In its simplest form, the Interface Function calls a utility function to create and initialize the interface and then, in the case of `popup_`, calls `VisualInterface_Manage()` to display the interface.

Note: You can specify whether the Interface Function is to be a `create_` function or a `popup_` function by choosing the default Interface Function Type from the Project Window's Options menu. Alternately, set the interfaceFunctionType resource in your Application Defaults. This resource can have the value `create` or `popup`. Changing this option will not affect previously created interfaces.

Linking Interfaces Together Using Instances

Another way to link interfaces together is to use instances. The advantage of using instances is that they will be created when the application is initialized. The disadvantage is that the programmer does not control when the interfaces are created or destroyed, and thus has less control over the amount of memory being consumed by the application.

Passing the Parent to the Interface Function

All generated interface functions declare the parameter `UxParent` of type `swidget`. The actual value passed to the interface function specifies the parent of the interface. You can pass the constant value `NO_PARENT` to the interface function if you want to create a top-level object.

Styles of Handling Interfaces

The create and popup functions allow a range of different styles for handling interfaces. The one to use will depend upon the type of application that you are creating and individual preference. The following styles can be implemented:

- Pop-up all interfaces as they are needed, and never pop them down. This is the simplest form of the connection of interfaces. It is not used frequently in applications but is included here as an example of the minimum required to accomplish the task.
- Create all interfaces at initialization time and pop them up and down as needed.
- Pop-up an interface every time it is needed and destroy it after use.
- Pop-up an interface when it is first needed and pop it down and up multiple times afterwards.

Each of these styles is explained in the sections that follow.

Regardless of the style chosen, in order for an Interface Function to be called it must be known to the calling interface. This is achieved with the following declaration in the calling interface.

Consider an application that has a Bulletin Board (`bulletinBoard1`) containing a Push Button that will pop-up interface `form1` when activated. In interface `bulletinBoard1` use the Declaration Editor and enter the following line in the Includes, Defines, and Global Variables section:

```
   extern swidget popup_form1
             UXPROTO((swidget UxParent, other parameters));
```

or

```
   extern swidget create_form1
             UXPROTO((swidget UxParent, other parameters));
```

depending on the style chosen.

Note: UXPROTO() is a macro defined by UIM/X that allows the declaration of a function to be compatible with K&R compilers as well as ANSI C and C++ compilers.

Create and Popup Interfaces as Needed

If, for example, you would like to pop up an interface (`form1`) when the user clicks on a button (`pushButton1`) in your application, and keep that interface on the screen, use the `popup_` function for `form1`. Then add the following lines of code to the activate callback for `pushButton1`.

```
/* Create and display the interface */
popup_form1(NO_PARENT);
```

Note that this is this simplest style of connection in that no attempt is made to remove the interface when it is no longer necessary nor to redisplay it when it is needed again.

Creating All Interfaces at Initialization Time

The second style is to create all interfaces at the beginning of the program, and then to pop them up and down as they are needed. In this style, the `create` function is most useful. In the section of the main program that begins with the comment *Initialization Code* you would add a line (using the Program Layout Editor) such as the following:

```
    sw1 = create_form1(NO_PARENT);
```

for each interface. The create function creates and initializes the interface, but does not display it. The swidget variable `sw1` is used as a handle to the interface. Later, when you want to display the interface, the following function call is used:

```
    VisualInterface_Manage(sw1, &UxEnv);
```

The method `VisualInterface_Manage()` works on all interfaces, including interfaces that are subclasses and instances of other interfaces. Whenever you *first* pop up a subclass or instance of another interface, you must use `VisualInterface_Manage()`.

Otherwise, if an interface is not a subclass or an instance, you can pop it up by calling the function `UxPopupInterface()`:

```
   UxPopupInterface(sw1, no_grab);
```

Note that after calling `VisualInterface_Manage()`, you can use `UxPopdownInterface()` and `UxPopupInterface()` to pop the interface down and up (even if the interface is a subclass or instance).

Similarly, use the following call to pop down the interface:

```
   UxPopdownInterface(sw1);
```

Note: If you choose to create all interfaces at initialization time, you should consider using instances as an alternative. See *"Linking Interfaces Together Using Instances"* on page 76 for more information.

Create an Interface When It Is Needed then Destroy it

The third style is to create an interface only when it is popped up for the first time. Here the popup function is more useful since it includes a call to `UxPopupInterface()` internally. To create, initialize, and display an interface, you would call:

```
   sw2 = popup_form2(NO_PARENT);
```

The function call:

```
   UxDestroyInterface(sw2);
```

can be used to pop down and destroy an interface after it has served its purpose.

Popping an Interface Up and Down Multiple Times

The fourth style is to create the interface when it is first needed, and pop it down and back up whenever it is subsequently needed. There are two ways of doing this. First, the popup function can be used to create and pop it up the first time

```
sw3 = popup_form3(NO_PARENT);
```

To pop it down, use:

```
UxPopdownInterface(sw3);
```

The next time the interface is to be displayed, it is not necessary to recreate it, so the following call can be used:

```
UxPopupInterface(sw3, no_grab);
```

Another approach for this style is to modify the popup function so that it checks whether the interface has been created already. In this case, the popup function can be used every time the interface is to be displayed. This can be achieved using the UIM/X Declaration Editor. By default, the popup function has the following structure:

```
swidget popup_form4(swidget UxParent)
{
    swidget rtrn;
    /*   Initial code   */

    rtrn = build_form4();
    /* Final code */

    VisualInterface_Manage(rtrn, &UxEnv);
    return (rtrn);
}
```

To modify this function, declare a static swidget variable and use it to save the result of the `build_form4()` function, which creates and initializes the interface:

```
swidget popup_form4(swidget UxParent)
{
    swidget rtrn;

    /* Initial code       */
    static swidget save_form = NULL;

    if (save_form == NULL) {
        rtrn = build_form4();

    /* Final code         */
        save_form = rtrn;
    }
    else
        rtrn = save_form;

    VisualInterface_Manage(rtrn, &UxEnv);
    return (rtrn);
}
```

It is clear that the `build_form4()` function will only be called the first time `popup_form4()` is called, so the following pairs of calls can be used to pop up and pop down the interface:

```
sw4 = popup_form4(NO_PARENT);
```

```
UxPopdownInterface(sw4);
```

Building Parametric Interfaces

7

Overview

The initial values of a property often depend on the state of the application. Considerations include what files have been loaded, what the user has previously done, other data sources, and the state of other interfaces. Furthermore, an interface might need to get its resource value from another object’s resource (like getting text from the Text widget). In such cases, the initial state of the interface is said to be *dynamic*—it changes each time a dialog box or an interface is popped up.

When building interfaces with a dynamic initial state, it is often convenient to design an interface whose initial state depends upon arguments passed to its Interface Function. This is particularly true when you want to have multiple instances of an interface with each one slightly different. Such an interface is called a parametric interface.

There are two steps to building a parametric interface. The first is to add arguments to the Interface Function. For a complete discussion of adding arguments see Chapter 6, "Editing Interface Code". The second, discussed here, is to use those arguments as expressions in the interface object properties.

Adding Arguments

The following code sample declares an Interface Function with an additional parameter, *label*. The parameter is used to write to the label of a Push Button in a drawing area each time the interface is created.

```
swidget popup_drawingArea1(swidget UxParent, char *label)
```

Putting Code in Properties

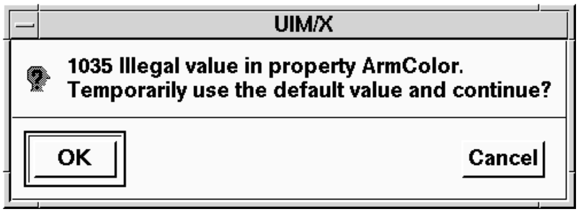
UIM/X allows you to enter an expression for the value of any property in the Property Editor. The only requirement of the expression is that it be of the correct type.

For example, many property resources, such as a Push Button's `LabelString` property resource, require a string value. In the Property Editor, you could enter a global variable declared `char*`, a function that takes arguments and returns a `char*`, or an argument to the popup function which is a `char*`.

The expressions are evaluated by the Interpreter when the object is created (or recreated) in UIM/X. As a result, by modifying the application state and creating or recreating the interface, the interface may be quite different. (Of course, the expression is kept as is in the generated code.)

Expressions That Don't Have a Legal Value

When an expression is entered in the Property Editor and the Apply button is clicked, the object is recreated. Often, however, the expression may not yet have a legal value because the application may not be built, running, or initialized. In such cases, a dialog box appears, similar to the one shown in Figure 7-1:



1035 Illegal value in property ArmColor.
Temporarily use the default value and continue?

Figure 7-1 Question Dialog

This message only appears when an expression is used and does not appear for constants. The message alerts you to the fact that there is an illegal value, or that the expression may have been incorrectly typed, although it would parse legally in the Interpreter. If an expression is correct and simply does not yet have a legal value, click on OK. Choosing Cancel will cancel the Apply. When OK is chosen, the object's default value for the property will be used instead. If an expression defined in the Declaration Editor is used as a resource, applying the Property Editor is not enough. Any modification to the expression definition will require applying the Declaration Editor.

Note that every time the object is created or recreated, the expression is evaluated. If it is recreated because of a Property Editor Apply and the value is still illegal, the dialog box will reappear. If it is recreated because the popup function was called or because the Recreate menu option was chosen, a message will simply appear in the Messages Area of the Project Window.

Note: You can avoid the illegal value message in the Property Editor by entering an expression that will always evaluate to a legal value, even if the variable is not yet initialized. For a property expecting a character string, the following would be legal:

```
(label ? label : "No Label")
```

For a property expecting a positive integer the same type of expression would also work:

(value ? value : 100)

Putting Arguments in Properties

Once you have declared an argument to the Interface Function in the Declaration Editor and clicked on Apply, you may use it to parametrically define properties. An argument to the Interface Function may be used in any property of any object in the interface. For instance, suppose that instead of a label string in quotation marks, the `LabelString` property of the Push Button contains the following variable name (note the absence of a semicolon):

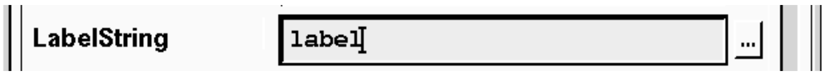


Figure 7-2 Putting an Argument in the LabelString Property

By calling the Interface Function with a string as an argument, for example:

```
popup_drawingArea1(NO_PARENT, "This will appear
                              on the pushbutton.");
```

the interface will be parametrically created—the label on the Push Button will be the one provided by the Interface Function.

Multiple Copies of Interfaces

In compiled code or code loaded into the Interpreter, (although not for an interactively created interface), you may call an Interface Function several times and create many copies of an interface, all of which may be visible simultaneously. An example of this is the Property Editor, many of which may be visible at a time. Each is parametrically created with a different object as the argument to the Interface Function. When you create multiple copies of an interface in this manner, each copy keeps some context-specific information. For example, if there were a Push Button and a Frame object in the interface and the activate callback for the Push Button contained the code:

```
    {
        UxPutBackground(frame1, "red");
    }
```

the callback code for each copy of the interface created would be identical, yet each one would set a different object's background to red (i.e., it would only set the background color of the frame in its own interface). This is accomplished through the context structure.

The Contents of the Context Structure

The context structure tells UIM/X which interface to act upon when a callback function or action function is called.

Each interface has a context structure, allocated when the interface is created. The context structure stores the objects of local scope, context-specific variables, and the arguments passed to the Interface Function.

The type of the context structure is `_UxC`*interfaceName*, where *interfaceName* is the name of the top-level swidget of the interface. Each time the Interface Function is called, a new copy of the context structure is allocated to hold the variables for the new occurrence of the interface. The name of this context structure is `Ux`*InterfaceName*`Context`, where *InterfaceName* is the (capitalized) name of the top-level swidget of the interface.

As an example, consider an interface with three swidgets (all of local scope):

`drawingArea1`	`frame1`	`pushButton1`

that has the following Interface Function declared:

```
swidget popup_drawingArea1(swidget UxParent, char *label)
```

and has two variables declared in the Interface Specific section of the Declaration Editor:

```
    int         IV1;
    float       IV2;
```

The context structure for this interface would be declared as:

```
typedef struct
{
swidget        UxdrawingArea1;
swidget        Uxframe1;
swidget        UxpushButton1;
int            UxIV1;
float          UxIV2;
swidget        UxUxParent;
char           *Uxlabel;
} _UxCdrawingArea1;
```

The generated C code would contain a variable declared as:

```
static _UxCdrawingArea1 *UxDrawingArea1Context;
```

This variable is used to store a pointer to the context structure that is currently in use.

A series of macro definitions makes it possible to refer to the elements of the context structure by using the simple variable names:

```
#define                UxDrawingArea1Context->UxdrawingArea1
drawingArea1           ;

#define frame1         UxDrawingArea1Context->Uxframe1;

#define pushButton1    UxDrawingArea1Context->UxpushButton1;

#define IV1            UxDrawingArea1Context->UxIV1;

#define IV2            UxDrawingArea1Context->UxIV2;

#define label          UxDrawingArea1Context->Uxlabel;

#define UxParent       UxDrawingArea1Context->UxUxParent;
```

Note: In generated C++ code, such macro definitions are not needed since all interface-specific variables and swidgets are within the scope of the C++ class representing the interface (i.e. C++ automatically handles the context by means of the implicit `this` pointer.

Note that the existence of these macros in the generated C code makes it impossible to declare a local variable, label, or any other identifier in your callback, initial, final, or auxiliary code having the same name as one of the swidgets, context-specific variables, or arguments passed to the Interface Function. If you do this, the macro expansion results in a syntax error.

Furthermore, you must watch out for any hidden uses of these identifiers, such as in library macros which might expand into expressions containing them.

For example, if you use the X11 function DefaultColormap in your interface, you will not be able to have a context-specific variable named “cmap”, because DefaultColormap is actually a macro which expands to reference a member called “cmap” of an X11 data structure.

Upon entering a callback or action function, the current context is saved in a local variable and then the relevant context is determined from the `UxWidget` parameter and made the current context. At the end of the callback or action function, the current context is restored from the local variable where it was saved.

For example, here is the C code generated for the activate callback for `pushButton1`:

```
static void activateCB_pushButton1(Widget UxWidget,
                                   XtPointer UxClientData,
                                   XtPointer
UxCallbackArg)
{
    _UxCdrawingArea1 *UxSaveCtx, *UxContext;
    swidget            UxThisWidget;

    UxThisWidget = UxWidgetToSwidget(UxWidget);
    UxSaveCtx = UxDrawingArea1Context;
    UxDrawingArea1Context = UxContext =
                    (_UxCdrawingArea1 *) UxGetContext
                    (UxThisWidget);
    {
                    /* YOUR CODE GOES HERE */
    }
    UxDrawingArea1Context = UxSaveCtx;
}
```

Note: Note that if you set the scope of a swidget to be global, then it will not be a part of the context structure. Instead, the swidget value is stored in a global variable. The use of swidgets of global scope is not recommended if there will be multiple copies of the interface.

Building Reusable Components

UIM/X provides the ability to create reusable components that deliver all the functions necessary for most applications, while remaining easy to build. Reusable components follow from multiple parametric instances of top-level objects. They add the capability of creating multiple parametric instances of components (object hierarchies) that can be children of other objects. Components are described in Chapter 9, "Working with Components, Subclasses, and Instances".

Parametric Instances

Using the Parent as an Argument to the Interface Function

The Parent field of top-level objects is an expression. When an implicit shell is created programmatically through a call to its Interface Function with an object as a parent rather than the constant `NO_PARENT`, UIM/X strips off the implicit shell and creates the interface as a child of the object in its Parent field. The parent is normally passed as an argument to the Interface Function, allowing dynamic control over the parent of each instance.

For example, a bulletin board could become a parametric component child of a Form object. In the Declaration Editor's Interface Function, you would enter:

```
   swidget create_bulletinBoard1(swidget UxParent)
```

In the Property Editor of the bulletin board, you would enter the variable `UxParent` in the Parent field in the Declaration properties.

You would select the Form interface icon in the Interfaces area of the Project Window, open the Interpreter and switch to Test Mode. In the Interpreter, you would choose Module⇒Selected Interface and enter the following:

```
    extern swidget create_bulletinBoard1();
    create_bulletinBoard1(form1);
```

Finally, you would highlight `create_bulletinBoard1(form1)` and choose Interpret⇒Evaluate.

`bulletinBoard1` becomes a child of `form1`, rather than being a top-level object.

Alternatively, you might create a Push Button in the Form object and enter the same functions placed in the Interpreter as a callback of the Push Button.

What Is a Dynamically Created Object?

A dynamically created object is an object that has been created in code through calls to the Ux Convenience Library. A dynamically created object is analogous to dynamically allocated memory (as opposed to statically allocated memory). If you return to the initial state of the program, the dynamically allocated memory no longer exists. Because dynamically created objects are not part of the initial state of the interface (for example, they were created from a callback), they will be lost when you re-create the interface.

Therefore, when you attempt an interactive operation on an object hierarchy that contains a reusable component, a dialog informs you that the operation will destroy all dynamically created objects in the interface. The dialog prompts you to confirm that you wish to continue.

Building Reusable Interface Components 8

Overview

Every graphical user interface (GUI) contains groups of related objects that work together. Often the same groups of objects appear over and over again in a GUI. A text entry area that combines a Label and a Text Field is a familiar example of a frequently used group of objects.

When a specific group of objects is so useful, it's convenient to treat it as a higher-level user interface component, rather than as a collection of primitives. This makes it easier to reuse the group of objects. Instead of having to create and properly assemble a collection of objects, you just have to create a single component.

Consistency is another advantage of packaging a collection of objects as a single component. Each time you create the component, it looks and behaves the same way. Such components can be used to establish and promote user interface standards.

The Motif user interface toolkit provides a number of such components. Menus, menu bars, dialogs, and main windows are examples of pre-configured groups of objects that can be created as single user interface components.

UIM/X provides a set of object-oriented features to help you build your own reusable components, whether you program in C or in C++. Taken together, these features form the UIM/X Component model. This model provides a visual metaphor for building and reusing classes of user interface components.

Note: From the C++ perspective, a component is a C++ class. So why not call a class a class? Because in UIM/X, you can choose to implement a class in either C or C++. For this reason, this document uses the more general term *component*.

The first part of this chapter describes the UIM/X Component model and its different elements. It uses an example project to explain how to use these elements to build reusable components. You can find this project in *uimx_directory*`/contrib/LabelTextField`. The second half of the chapter explains the UIM/X Component Model from the perspective of the generated code.

Understanding the UIM/X Component Model

The UIM/X Component model supports object-oriented programming in either C or C++. It provides a mechanism for building class hierarchies with inheritance, encapsulation, and polymorphism.

The key to understanding the UIM/X Component model is the observation that UIM/X interfaces are really *classes*. In UIM/X, anything you see on the desktop is a class. Therefore, in the `LabelTextField` project, the Form object and its children, the Label and the Text Field, combine to form a class.

These interface classes are called *Components*. Components follow the Common Object Broker Request Architecture (CORBA) standard for C and C++ classes. The CORBA standard specifies how to build classes in different languages, such as C, C++, or Smalltalk, so they can all interact.

To build a class hierarchy, you derive new classes from a Component. These derived classes are called *Subclasses*. *Methods* are the operations that you can perform on the objects of these classes. (In the UIM/X Component model, objects are called *Instances*.)

Building Components

In UIM/X, you are always building Components, because anything you put on the desktop is really a Component. From a shell to a Push Button to a full-blown interface, anything you see with window manager decorations is a Component. The only question is whether or not you intend to reuse the Component.

For example, suppose that after building a sophisticated interface, you look at it and realize that one of its elements can be reused. UIM/X provides a simple way to convert such an interface element into a Component.

All you have to do is drag the element out of the interface and onto the desktop. This converts the element into a Component. When you do this, UIM/X also gives you the option of replacing the element (in the original parent) with an Instance of the new Component.

Creating Instances

You reuse a Component by creating an Instance. You create Instances just as you create swidgets, by drawing them. Figure shows several Instances of a simple LabelText Component drawn on another interface. The LabelText Component combines a Label and a Text Field on a Form to make up a reusable text entry area.

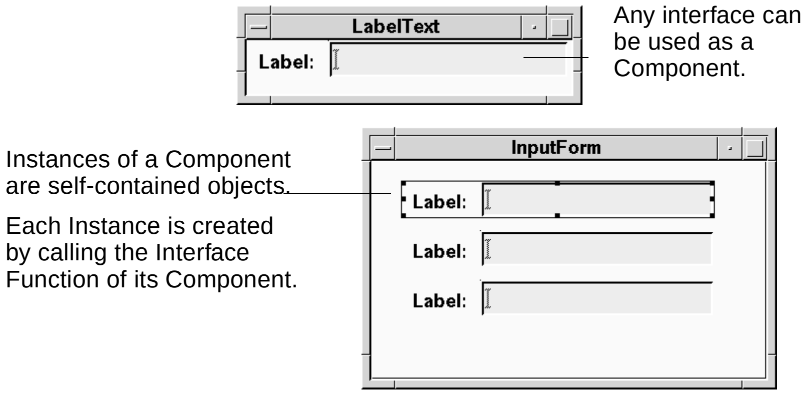

LabelText
Label:
Any interface can be used as a Component.
InputForm
Label:
Label:
Label:
Instances of a Component are self-contained objects.
Each Instance is created by calling the Interface Function of its Component.

Figure 8-1 Instances of a Component

The standard interface in creating an instance of a Component is the Component’s Interface Function. This function creates an Instance of the Component and returns a swidget that represents the top-level object of that Instance. When code is generated for an interface, each Instance you draw in UIM/X inserts a call to the Component’s Interface Function in the generated code:

```
// Creation of LabelTextInstance1
LabelTextInstance1 = create_LabelText( InputForm, 0 );
```

In this example, the function `create_LabelText()` is the Interface Function for the LabelText Component. It creates an Instance as a child of the swidget `InputForm`.

In C++, there are alternatives for creating an Instance of a Component. The C++ class generated for a Component includes the following:

- a default (parameterless) constructor
- a parametered constructor
- a `CreateSwidget()` member function

Both the parametered constructor and the `CreateSwidget()` member function accept the same arguments as the Component's Interface Function, and both achieve the same effect. That is, they build the swidget tree of the Instance, set all initial properties, and register all callbacks. And as with the Interface Function, they initiate execution of the Initial Code and Final Code entered in the Declaration Editor.

As a result, the following are all valid ways of creating an Instance of a Component in C++:

- Using the Component's Interface Function and dealing with the Instance as a swidget:

```
swidget LabelTextInstance1;
LabelTextInstance1 = create_LabelText( InputForm, 0 );
```

- Using the Component's default constructor and calling the `CreateSwidget()` member function:

```
_UxCLabelText LabelTextInstance1;
LabelTextInstance1.CreateSwidget( InputForm, 0 );
```

or:

```
_UxCLabelText *LabelTextInstance1;
LabelTextInstance1 = new _UxCLabelText;
LabelTextInstance1->CreateSwidget( InputForm, 0 );
```

- Using the Component's parametered constructor:

```
_UxCLabelText LabelTextInstance1( InputForm, 0 );
```

or:

```
   _UxCLabelText *LabelTextInstance1;
   LabelTextInstance1 = new _UxCLabelText( InputForm, 0 );
```

When C++ code is generated for an interface, each Instance you draw in UIM/X is represented by an object of the Component's class, and a call to the Component's `CreateSwidget()` member function is used to create the Instance:

```
   // Creation of LabelTextInstance1
   LabelTextInstance1.CreateSwidget( InputForm, 0 );
```

Note: These C++ mechanisms for creating Instances are not available at design time. Thus, if any of the mechanisms are used in code that is subject to being interpreted during design time, they must be protected by an `#ifndef DESIGN_TIME` block. Furthermore, these mechanisms are unavailable if Ux Convenience Library C++ Bindings are disabled. See *"Generating C++ Code without C++ Bindings"* on page 117 for more information.

Because the Interface Function of a Component contains calls to the Interface Functions of the Instances within the Component, you cannot create an Instance as a child of its Component. This would result in an infinite series of calls to the Component Interface Function, because the Interface Function would have to call itself to create the child Instance. Creating an Instance as a child of its Component is analogous to declaring a class C that contains an object of class C, which is something you cannot do in C++.

Note: You *can* use Instances to build a new Component. One Component can contain Instances of another Component, in the same way that a C++ class can include objects of other classes as members.

Creating Instances with Explicit Shells

When you create a Component, UIM/X automatically adds a shell swidget. This shell swidget is called an *implicit shell*, because it allows you to create objects on the desktop without explicitly creating a shell swidget. (By default, this implicit shell is a TopLevelShell.) When you create an Instance, the implicit shell is removed. It is not part of the Component.

If you want a shell to be part of a Component, you have to create an *explicit shell*. To do this, you create a shell swidget as you would any other swidget. There's nothing special about an explicit shell, other than the fact that you have to explicitly create it.

However, when a Component includes an explicit shell, its Instances are always top-level. If you create an Instance with an explicit shell as a child on another interface, the instance can be seen in the Browser but is not visible on the interface, nor does it have an icon in the Interfaces area of the Project Window.

Defining Methods

A Component has a public interface that consists of an Interface Function and a set of methods. The Interface Function is the function that creates an Instance of the Component. Methods are the operations that can be applied to Instances of the Component.

In generated C++ code, methods are implemented as member functions. In UIM/X and in generated C code, methods are implemented by a method dispatcher that stores and retrieves function pointers in a lookup table.

Methods are an important part of the UIM/X Component model. They are part of the encapsulation provided by a Component, because they separate code that uses a Component from the code that implements the Component. Interface code that uses a Component does so by invoking its methods. The internal state of a Component is never directly accessed, except by its methods.

In addition, methods allow you to define *polymorphic* operations—operations that behave differently on different classes of a Component. (And they let you do this in either C or C++.)

Invoking Methods

You invoke a method on an Instance using a method macro:

```
LabelText_Validate( LabelText, other_args, &UxEnv )
```

The macro name is formed by combining the name of the Component and the name of the Method. The naming and calling conventions of the method macros follow the CORBA specifications. UIM/X gives you a choice between Corba 1.1, Corba 2.0, or no Corba.

The method macros are defined both in UIM/X and in the generated code. For example, you can use method macros as initial value expressions in the Property Editor. The method macros map the method call to the appropriate code. In C++, the method macros expand to calls to member functions of the interface class. See *"Methods and Method Macros"* on page 113 for more information.

The first argument is the *top-level swidget* of the Instance. If you choose Corba 1.1 or Corba 2.0 compliancy, the argument list also contains a pointer to an *Environment* structure (as the second argument if you choose Corba 1.1, and as the last argument if you choose Corba 2.0). This *Environment* pointer is required by CORBA. Its intended use is the return of exception information. While UIM/X does not raise request broker exceptions, this argument allows you to handle exceptions if you integrate your applications with a CORBA-compliant object request broker.

In the past, you invoked a method by passing any swidget in an Instance as the first argument to the method macro. While UIM/X still supports this way of doing things, it now provides a stricter mode of method invocation.

The UIM/X resource `UxStrictMethodInvocation.set` controls whether or not UIM/X enforces the strict mode of method invocation. By default, strict mode is turned off. To run UIM/X in strict mode, set `UxStrictMethodInvocation.set` to `True`.

In strict mode, you can invoke methods *only* on the top-level swidget of an Instance. Since an Instance encapsulates a swidget tree, it doesn't make sense to invoke methods on any swidget in the tree. Methods apply to the Instance itself, not to the individual swidgets it encapsulates. In strict mode, UIM/X considers the top-level swidget as the only swidget that can represent the Instance.

To understand how the top-level swidget can represent an Instance, consider that the function that creates an Instance (the Interface Function) returns the top-level swidget. And when you destroy the top-level swidget, you destroy the entire swidget tree. So in the UIM/X model, where methods have to work in C or C++, the top-level swidget really does represent the Instance.

When UIM/X is in strict mode, method invocations that do not use the top-level swidget have no effect. UIM/X will not be able to find the function that implements the method, and will call a function that does nothing.

Note: Do not run UIM/X in strict mode if you have integrated C++ Components that do not follow the strict mode rules. When UIM/X is in strict mode, method invocations that do not follow the strict mode rules can cause UIM/X to exit during testing.

Understanding Methods and Polymorphism

The method macros provide a CORBA-compliant mechanism for invoking operations that depend on the type of object. For example, if class D is a subclass of class B, calling:

```
  B_mymethod( object, args, pEnv );
```

on an object of class D ultimately calls the version of the method defined by class D. The class of the object passed to the method determines the version of the method invoked. If a subclass does not override the method, the superclass version is invoked.

This is conceptually similar to C++, where the object would be declared as class B. Think of it as `object->mymethod(…)`, where you are stating (within the call, since C doesn't have strong typing) that it is an object of class D.

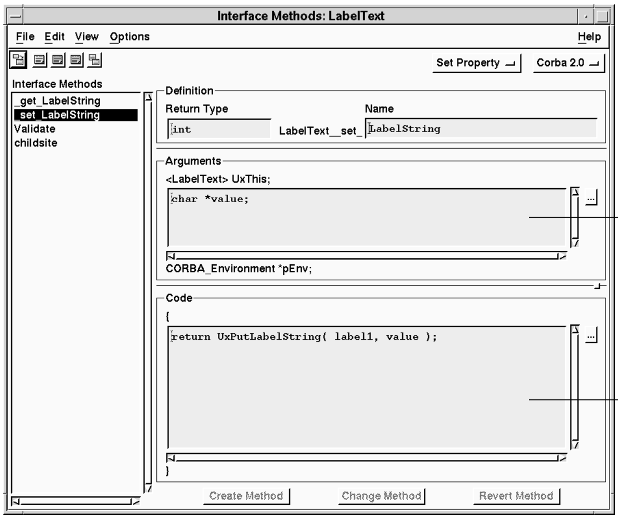
This calling convention works regardless of whether you are working in C or in C++. As a result, you can do object-oriented programming in C now. To switch to C++, you simply change a code generation option and all your code still works.

For an example of how to use polymorphism, see *"Overriding Callbacks in Subclasses"* on page 105.

Adding Instance Properties

A Component encapsulates a collection of swidgets and Instances. By default, the properties of the swidgets and Instances in a Component are private details that cannot be edited. But they can be made public by defining a pair of methods on the Component—one to set the property value and one to retrieve the property value.

These methods are known as *property accessors*. This modeling of properties as a pair of methods follows the CORBA specifications. Figure shows an example of a property accessor method. See *"Defining Property Accessors"* on page 126 for details on defining a property accessor.



Interface Methods: LabelText

File Edit View Options Help

Set Property Corba 2.0

Interface Methods
_get_LabelString
_set_LabelString
Validate
childsite

Definition
Return Type: int
Name: LabelText__set_ LabelString

Arguments
<LabelText> UxThis;

```
char *value;
```

CORBA_Environment *pEnv;

Code
{

```
return UxPutLabelString( label1, value );
```

}

Create Method Change Method Revert Method

A set accessor method takes one argument: the property value.

The body of the method contains the code necessary to set the property.

Figure 8-2 Property Accessor Method

These properties appear in the Specific category when you load an Instance into the Property Editor. You can even install option menus and specialized resource editors for these properties.

Note: Property accessors let you add properties of *any type*, not just Motif properties. The internal representation of an accessor property is private to the Component, and may be a swidget property, an interface-specific variable, or some other application-level attribute.

For example, suppose Component A contained an Instance of Component B. How would you expose the properties of this Instance?

The answer is, of course, by defining property accessors on Component A. But in these property accessors you would not call `UxPutProperty()` or `UxGetProperty()`, but rather the property accessors defined by Component B.

You can also add properties by adding arguments to the Component's Interface Function. These arguments become Core properties of an Instance.

Interface Function argument properties are like Motif properties whose access is C (you can set them only at Creation). Accessor properties are like Motif properties whose access is SG (you can Set or Get them after the Instance is created).

If a property must be set before an Instance is created, add an argument to the Interface Function. Otherwise use property accessors to define the property. Property accessors offer a number of advantages over Interface Function arguments:

- The property is inherited by Subclasses of the Component.

  In contrast, when you add a property by adding an argument to the Interface Function, the property is not inherited. Instances of a Subclass do not have the same property unless you explicitly add the same argument to the Subclass Interface Function.
- You can use the property accessors to set and retrieve the property value after an Instance is created.
- You don't have to supply a default value when an Instance is created.

Subclassing Components

You can derive a new class from a Component. The derived class, called a *Subclass*, inherits the structure and behavior of the Component. To create a Subclass, you create a top-level Instance. When you create an Instance of a Component on the desktop, UIM/X understands that you want to subclass the Component. Note that you can also derive new classes from a Subclass.

Figure shows an example of a Subclass created in UIM/X. Note that a Subclass can add its own swidgets to the swidget tree represented by the Component. See *"Building Manager Components"* on page 106 for more information.

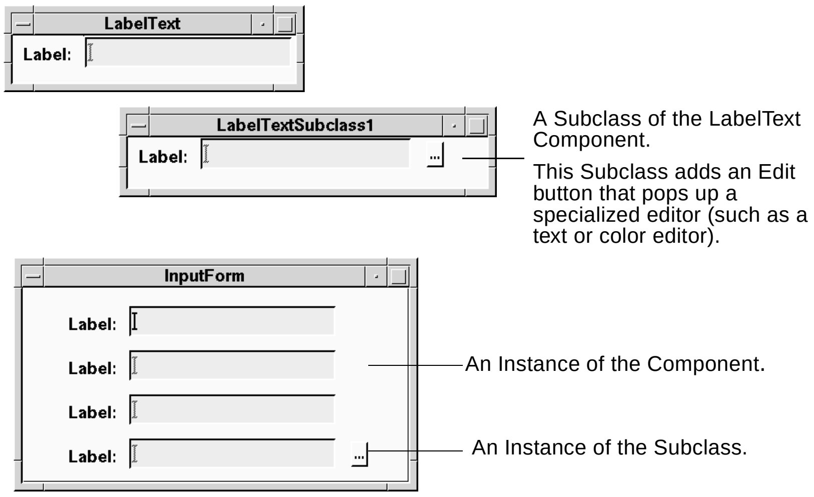

LabelText
Label:
LabelTextSubclass1
Label:
A Subclass of the LabelText Component.
This Subclass adds an Edit button that pops up a specialized editor (such as a text or color editor).
InputForm
Label:
Label:
Label:
Label:
An Instance of the Component.
An Instance of the Subclass.

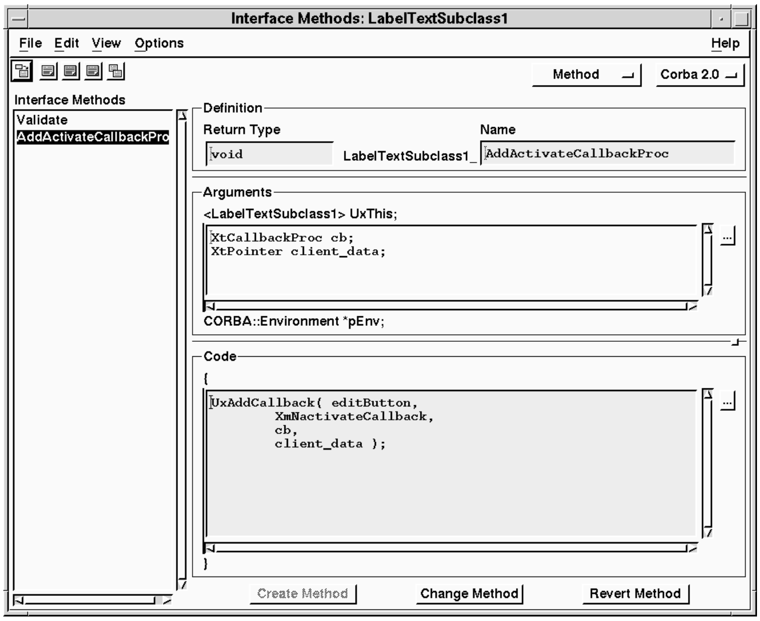
Figure 8-3 Subclassing a Component

Subclasses inherit the methods of the Component and can override them by defining a method with the same name, return type, and arguments. Because a Subclass inherits the Component methods, the Subclass inherits the Component properties (since properties are implemented as accessor methods).

By combining methods and Subclasses, you can build specialized versions of a Component. A Subclass can override an inherited method to provide behavior specific to itself. For example, suppose the LabelText Component defined a Validate method. A Subclass of the LabelText component could define its own version of the Validate method and restrict input to numerical values.

Adding Behavior Properties to Instances

If you want Instances to have Behavior properties that appear in the Property Editor, you can expose callbacks as Instance properties. To do this, you define a special method (called a *callback accessor*) on the Component. Figure shows an example of a callback accessor. See *"Adding Callbacks to Instances"* on page 129 for details on defining a callback accessor.



Return Type: void
LabelTextSubclass1_ AddActivateCallbackProc

<LabelTextSubclass1> UxThis;

```
XtCallbackProc cb;
XtPointer client_data;
```

CORBA::Environment *pEnv;

```
{
UxAddCallback( editButton,
        XmNactivateCallback,
        cb,
        client_data );
}
```

Figure 8-4 Callback Accessor Method

Using callback accessors, you can expose a callback as a function-pointer valued property that can be set in each Instance. These properties appear in the Behavior category of the Property Editor and are set using the Callback Editor.

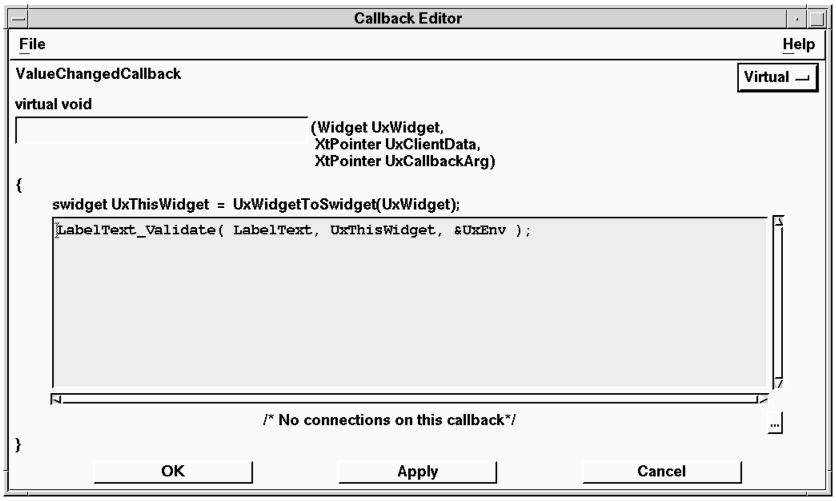
Callback accessors are not restricted to Motif callbacks. You can use them to add Behavior properties for events and messages generated by other systems. In a database application, for example, you could add a databaseChanged callback. The only restriction is that the procedure must have the signature of a standard Xt callback function.

Overriding Callbacks in Subclasses

Sometimes you want all Instances of a Component to have the same behavior, but you also want to be able to override this behavior in Subclasses. To do this, you exploit the fact that UIM/X methods are polymorphic.

First, you define a method that implements the callback behavior. Then you invoke this method from the callback. Subclasses can then override the callback behavior by overriding the method.

For example, the LabelText Component contains a TextField swidget. Suppose you wanted the valueChangedCallback of the TextField swidget to invoke a Validate method defined by the Component. To do this, you would write a callback like the one shown in Figure 8-5.



```
virtual void
                    (Widget UxWidget,
                     XtPointer UxClientData,
                     XtPointer UxCallbackArg)
{
    swidget UxThisWidget = UxWidgetToSwidget(UxWidget);
    LabelText_Validate( LabelText, UxThisWidget, &UxEnv );
    /* No connections on this callback*/
}
```

Figure 8-5 Invoking a Method in a Callback

In a Subclass of LabelText, the same callback function is called, but the Subclass version of the Validate method is invoked, not the LabelText version. This is because it is not the name of the macro that is important, but the class of the object (the first argument) passed to the macro.

Note that the first argument to the method macro is `LabelText`, the top-level swidget of the Instance, not `UxThisWidget`. This follows the strict mode rules for method invocation—see *"Invoking Methods"* on page 99 for more information.

If you examine the generated method macro for the Validate method, you will see that the first argument serves as an implicit self-reference to the Instance for which the method was invoked:

```
#ifndef LabelText_Validate
#define LabelText_Validate( UxThis, pEnv ) \
(((_UxCLabelText *)UxGetContext(UxThis))->Validate( pEnv
                 ))
#endif
```

The function `UxGetContext()` will return the `this` pointer for the object that contains the swidget `LabelText`. (See the discussion of `UxGetContext()` in *"Methods and Method Macros"* on page 113.)

Building Manager Components

The UIM/X Component model provides a way to build manager Components, whose Instances can accept swidgets and other Instances as children.

You build a manager Component by defining a special `childSite` method that returns a swidget (or Instance) that can be used as a parent. For example, if a Component consisted of a Label and TextField on a RowColumn, the `childSite` method would return the RowColumn swidget. Note that when a Component has a `childSite` method, Subclasses can add new swidgets as children.

For more information about the `childSite` method, see *"The Base Interface Class"* on page 107.

Understanding the Generated C++ Code

When you generate C++ code for a Component, you get a header file that contains the class definition and a source file that contains the class implementation. Anywhere an Instance of the Component is used, a call to the Component Interface Function appears.

You can take two views of the UIM/X Component model. One view is from the perspective of the visual development environment provided by UIM/X. The other is from the perspective of the generated code. Considering these two different but complementary views will help you to understand and use the features of the UIM/X Component model.

The Base Interface Class

A good place to start is with a brief look at the class from which all generated Component classes are derived. This class, called UxBase, provides functionality common to all Components, and is defined in *uimx_directory*`/include/UxBase.h`:

```
class UxBase
{
     protected:
         swidget UxThis;

     public:
         UXINLINE UxBase();
         UXINLINE ~UxBase();

         /* type conversion to swidget */
         UXINLINE operator swidget();

             .
             .
             .

};
```

The UxInterface and UxVisualInterface classes are derived from UxBase, and are defined in *uimx_directory*`/include/UxInterf.h`:

```
class UxInterface : public UxBase
{
      public:
          UXINLINE UxInterface();
          virtual ~UxInterface();
          UXINLINE Boolean auto_destroy() const;

          static void UxDestroyContextCB( Widget,
                                     XtPointer, XtPointer );

      protected:
          UXINLINE void auto_destroy(Boolean value);
          Boolean auto_destroy_flag;
};

class UxVisualInterface : public UxInterface
{
      public:
          virtual swidget childSite (Environment *pEnv);
          virtual swidget UxChildSite (swidget sw);
          virtual void _set_x (Environment*, int);
          virtual int _get_x (Environment*);
          virtual void _set_y (Environment*, int);
          virtual int _get_y (Environment*);
          virtual void _set_width (Environment*, int);
          virtual int _get_width (Environment*);
          virtual void _set_height (Environment*, int);
          virtual int _get_height (Environment*);
          virtual void Manage (Environment *pEnv =
               NULL);
          virtual void Unmanage (Environment *pEnv =
               NULL);
          virtual swidget get_UxThis();
};
```

The UxVisualInterface base class, helped by the classes from which it is derived, captures features common to all visual generated classes, including:

- A protected `UxThis` data member that holds the top-level swidget of the swidget tree encapsulated by the generated class. The `UxThis` member is set by the `_build()` member function, which creates the swidget tree.
- A `get_UxThis()` member function for retrieving the top-level swidget.
- A type-conversion operator allowing an implicit conversion to the top-level swidget.
- A virtual destructor.
- Member functions for setting and retrieving the x, y, width, and height properties of Instances. These member functions are defined in `UxInterf.cc`:

```
void UxVisualInterface::_set_x(Environment*, int x)
{
   XtVaSetValues(UxRealWidget(UxThis), XmNx, x, NULL);
}
```

- `Manage()` and `Unmanage()` member function for managing and unmanaging Instances.
- A `childSite()` member function that returns a swidget that can be used as a parent. The Interface version returns 0, which means that Instances of the class cannot accept children. A manager Component would override this member function with a version that returned a valid parent.
- The Interface class also defines a `UxChildSite()` member function that handles the case where one Instance contains another Instance, and the nested Instance is the intended parent. This member function is a recursive function that steps through a hierarchy of Instances, calling the `childSite()` method of each Instance.

  You use `UxChildSite()` when you want to create an object on an Instance:

```
   editButton = UxCreatePushButton( "editButton",

                  UxChildSite(_UxCLabelText::childSite(&Ux
                  Env)) );
```

Method macros for invoking these member functions are defined in `UxLib.h`. The actual implementation of the Interface class is contained in *uimx_directory*`/config/UxInterf.cc`. When you generate C++ code, the file `UxInterf.cc` is copied into the directory containing the generated files.

Generated Component Classes

Note: The discussion that follows assumes that Ux Convenience Library C++ Bindings are being used. If these bindings are disabled, Component classes are generated differently. For a description of how Component classes are generated when Ux Convenience Library C++ Bindings are not being used, see *"Generating C++ Code without C++ Bindings"* on page 117.

When you generate C++ code for a Component, the class representing the Component derives from UxVisualInterface (although if the Component is a Subclass of another Component this derivation will be indirect). The following is the class declaration for the Labeltext Component show in Figure 8-1.

```
class _UxCLabelText: public UxVisualInterface
{

// Generated Class Members

public:

   // Constructor and Destructor Functions

   _UxCLabelText( swidget UxParent, int anArg );
   inline _UxCLabelText() {}
   ~_UxCLabelText();

   // Interface Function

   swidget CreateSwidget( swidget UxParent, int anArg );

   //For backwards compatibility

   inline swidget _create_LabelText() { return UxThis; }

   // User Defined Methods

   virtual int Validate( swidget textWidget,
                  CORBA::Environment * pEnv );
   virtual char * _get_LabelString( CORBA::Environment *
                pEnv );
   virtual int _set_LabelString( char * value,
                  CORBA::Environment * pEnv );
   virtual swidget childSite( CORBA::Environment * pEnv );
protected:

   // Widgets in the interface

   UxForm LabelText;
   UxLabel label1;
   UxTextField textField1;

   // Arg List of creation function

   swidget UxParent;
   int anArg;

   // Callbacks and their wrappers

   virtual void valueChangedCB_textField1(Widget,
                 XtPointer,
                   XtPointer);
   static void Wrap_valueChangedCB_textField1(Widget,
                   XtPointer, XtPointer);

// User Defined Methods

private:

   _UxCLabelText &build();
   CPLUS_ADAPT_CONTEXT(_UxCLabelText)
   friend swidget create_LabelText(swidget UxParent, int
                 anArg);
} ;
```

Data Members

The representation of a class consists of the swidgets in the interface, interface-specific variables, and Interface Function arguments. These data members are declared in the protected portion of the class so only derived classes can access them.

Wherever possible, Ux Convenience Library C++ Bindings are used to declare the types of swidgets in the interface. The actual type `swidget` is used only where needed to maintain backwards compatibility with projects developed with earlier versions of UIM/X.

Because the Ux Convenience Library C++ Binding classes used to represent the swidgets of the interface also derive from the UxBase class, they inherit the type conversion operator that allows an implicit conversion to type `swidget`. This allows the objects of an interface to be used in all situations where historically only objects of type `swidget` were accepted (such as the first argument to all Ux Convenience Library functions).

The first argument to the Interface Function *must* be the parent swidget for the swidget tree represented by the Component. In the generated code, this first argument is always `UxParent`, so a generated class always has a `UxParent` data member.

Methods and Method Macros

Methods defined in the Method Editor become public member functions of the class. The class header file defines macros for invoking these methods. These method macros translate a method call on an Instance to a member function call:

```
#ifndef LabelText__get_LabelString
#define LabelText__get_LabelString( UxThis, pEnv ) \
   (((_UxCLabelText *) UxGetContext(UxThis))
                              ->_get_LabelString( pEnv ))
#endif
```

The method macros follow the Common Object Request Broker Architecture (CORBA) calling conventions.

A method macro is shorthand for `this->method()`. The call to `UxGetContext()` retrieves the `this` pointer for the C++ object on which the method is being invoked.

In the UIM/X model, the `this` pointer is stored in the swidget's X context (in the context manager database provided by Xlib). The functions `UxPutContext()` and `UxGetContext()` store and retrieve the `this` pointer in the context manager database.

The `this` pointer is stored when the swidgets are created by the `_build()` member function:

```
   // Creation of LabelText
   LabelText = UxCreateForm( "LabelText", UxParent );
   UxPutContext( LabelText, this );
   UxThis = LabelText;
```

Callbacks

In C++, the `this` pointer is passed as a hidden argument to all non-static member functions. This means that a non-static member function cannot be registered as a callback, because callbacks are C functions that do not accept the `this` pointer.

To handle this problem, UIM/X uses two member functions for each callback:

```
// Callbacks and their wrappers
virtual void valueChangedCB_textField1(Widget, XtPointer,
               XtPointer);
static void Wrap_valueChangedCB_textField1(Widget,
               XtPointer, XtPointer);
```

The static member function is registered as the callback. It calls the virtual member function to do the actual work. However, the static member function needs the `this` pointer so it can call the member function on the appropriate object.

In the Douglas Young model, the `this` pointer is passed as client data to the static member function. However, this use of the client data argument is restrictive. The purpose of the client data is to hold information specific to a callback (so you can share callbacks). For example, on a numeric keypad, all buttons might share the same callback, with the actual digit being stored in the client data.

In the UIM/X model, the `this` pointer is stored in the Xlib context manager database. The static member function retrieves the `this` pointer (by calling `UxGetContext()`) and uses it to call the appropriate member function:

```
   void _UxCLabelText :: Wrap_valueChangedCB_textField1(
                         Widget wgt,
                         XtPointer cd,
                         XtPointer cb)
   {
        _UxCLabelText *UxContext;
        Widget UxWidget = wgt;
        XtPointer UxClientData = cd;
        XtPointer UxCallbackArg = cb;
        swidget UxThisWidget;

        UxThisWidget = UxWidgetToSwidget( UxWidget );

// Retrieve the this pointer from the X context
        UxContext = (_UxCLabelText *)
                  UxGetContext(UxThisWidget);

// Call the member function that contains the callback code
        UxContext->valueChangedCB_textField1(UxWidget,
                         UxClientData, UxCallbackArg);
}
```

Instance Creation

In generated C++ code, the standard way of defining an Instance of a Component is to define an object of the class representing the Component. Such objects are defined as protected data members of the class that contains them.

```
class _UxCInputForm : public UxVisualInterface
{

              .
              .
              .

     protected:
         // Widgets in the interface
         UxForm                  InputForm;
         _UxCLabelText           LabelTextInstance1;
         _UxCLabeltext           LabelTextInstance2;
         _UxCLabelText           LabelTextInstance3;
         _UxCLabelTextSubclass1  LabelTextSubclassInstance1

              .
              .
              .

};
```

Defining an Instance of a Component in this way causes the default (parameterless) constructor of the Component's class to be called. Only later, in the `_build()` member functions of the Component class that contains the Instance, is the swidget tree of the Instance built. This is achieved by calling the Instance's `CreateSwidget()` member function.

```
// Creation of LabelTextInstance1
LabeltextInstance1.CreateSwidget( InputForm, 0 );
```

Instance Destructor

UIM/X generates a virtual destructor for every Component. The destructor is virtual because a Component is derived from a class with a Virtual destructor (UxInterface).

Because the Instances within a Component are defined as data members of the Component's class, the destructors of these Instances will automatically be called when the Component itself is destroyed. The destructor of each Instance will in turn destroy the swidget tree of the Instance.

By default, UIM/X does not generate destructors for Subclasses. Destroying an Instance of a Subclass ultimately calls the generated Component destructor. If you want a Subclass destructor, you can define it in the Declaration Editor. You can also add your own code to the generated Component destructor. See *"Adding Destructor Code"* on page 184 for more information.

Destroying the objects in an Instance is something of a problem for C++ destructors. Because destroying the top-level object destroys the entire object tree, it is possible for an Instance's objects to be destroyed before its destructor is called.

For example, suppose Instance A contains Instance B. When you call `delete` on Instance A, the destructor for Instance A is called first, then the destructor for Instance B. When the Instance A destructor destroys the objects in Instance A, it also destroys the objects in Instance B. If the Instance B destructor then tries to destroy its objects, it will be trying to destroy the objects a second time.

To handle this problem, the generated destructor checks the value of `UxThis`. If `UxThis` is `NULL`, the destructor knows that the objects have been destroyed.

Generating C++ Code without C++ Bindings

Instance Creation

When Ux Convenience Library C++ Bindings are disabled, generated C++ code, like generated C code uses swidgets to represent Instances rather than using actual objects of the Component's class. Thus, the Interface Function of a Component is used to create Instances of that Component.

You can still create Instances directly, bypassing the Component's Interface Function. However, the mechanisms for doing so are slightly different:

- There is only one constructor, that takes the same parameters as the Interface Function. However, unlike the parametered constructor available when Ux Convenience Library C++ Bindings are used, this constructor does not do any real work.
- There is no `CreateSwidget()` member function. Instead, an equivalent member function exists that has the same name as the Interface Function, except that it is prefixed with an underscore character.

It is important to note that even when Instances are created via their Component's Interface Functions and thereafter manipulated as swidgets, objects of the Component's class still come into play. The Interface Function of a Component dynamically creates an object of the Component's class, then calls the `CreateSwidget()` member function to build its swidget tree, set all initial properties, and register all callbacks.

```
swidget create_LabelText( swidget UxParent, int anArg )
{
     _UxCLabelText *theInterface =
             new _UxCLabelText( UxParent, anArg );
     return (theInterface->_create_LabelText());
}
```

The swidget returned by the Interface Function represents the top-level object of the Instance and has as its context a pointer to the dynamic interface itself. That is, the `this` pointer of the Instance is attached to the swidget via the Xlib context manager database. It is through this mechanism that Instances can be manipulated via their swidgets.

Instance Creation within the Constructor

It is possible to instruct UIM/X to generate the parametered constructor so that it automatically creates the swidgets. This allows Instances to be created in one step, in much the same way as is possible when using Ux Convenience Library C++ Bindings.

To generate a constructor that creates the swidgets, you have to add the `-z` option to the *uimx_directory*`/bin/uxcgen` command-line. The `-z` option forces `uxcgen` to put the call to the `_create_`*Component*`()` function in the constructor, not the Interface Function:

```
_UxCLabelText::_UxCLabelText( swidget UxParent, int anArg )
{
      this->UxParent = UxParent;
      this->anArg = anArg;

      this->_create_LabelText();
}
```

Code that already uses the Interface Function will still work, because even if you use the `-z` option, the Interface Function still returns the top-level swidget of the Instance. The difference is that instead of calling the `_create_`*Component*`()` function itself, the Interface Function relies on the constructor to make the call.

To use the `-z` option every time you generate code, you can modify the shell script *uimx_directory*`/bin/uxcgen.sh`. UIM/X calls this script whenever it has to generate code.

To add -z to the UIM/X code generation options:

1. In the file *uimx_directory*`/bin/uxcgen.sh`, add the `-z` option to the `uxc-gen` command-line: You can find these lines at the end of the script file.

```
      if [ "$cfile" != "" ]; then
          uxcgen $options -z -o $cfile $ifile;
      else
          uxcgen $options -z $ifile;
      fi
```

Note: Modifying the copy of `uxcgen.sh` in *uimx_directory*`/bin` affects all UIM/X users on the system.

To add -z to your personal code generation options:

1. Copy the contents of *uimx_directory*`/bin` to a local directory. You may want to copy only `uxcgen.sh`, and create symbolic links to the other files in *uimx_directory*`/bin`.
2. Modify `uxcgen.sh`.

3. Set the UIM/X resource `bindir` to point to the local directory:

```
        Uimx3_0.bindir: /bean/usr/douglas/bin
```

The resource `bindir` tells UIM/X where to find its executable and utilities. See *UIM/X Installation Guide* for more information on the UIM/X resource file and how to set application defaults for a single user.

Passing Arguments to Base Class Constructors

Passing a Subclass constructor argument to its base class constructor is accomplished through the Property Editor. In a Subclass, the base class constructor arguments appear as Core properties.

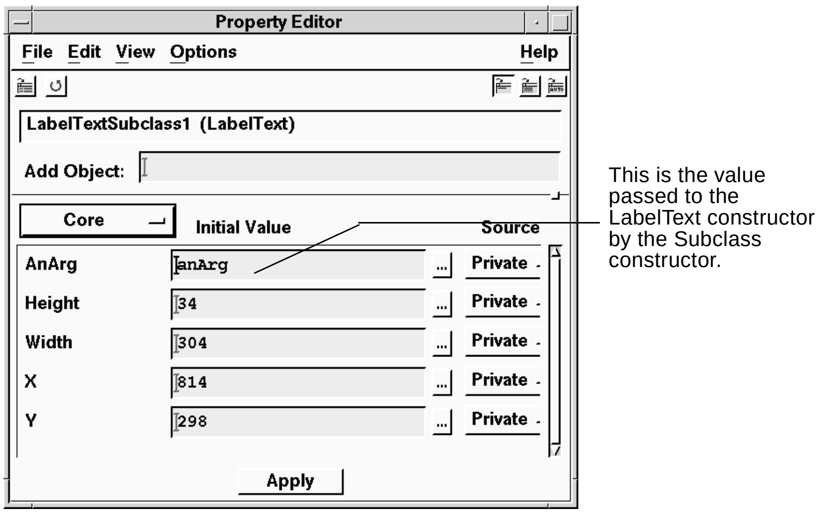
The values you enter in the Property Editor for these properties are the values passed to the base class constructor in the generated code. All you have to do is use the Subclass constructor arguments as the property values.

Note: In UIM/X, Interface Function arguments are also constructor arguments. In the generated code, the Interface Function passes its arguments to the constructor, so the two functions always have the same argument list.

For example, suppose you add the argument `anArg` to the Interface Function of the LabelText Component. This argument becomes a Core property in a Subclass of LabelText such as LabelTextSubclass1. In the generated code, the Initial Value entered in the Property Editor for the anArg property is passed to the LabelText constructor:

```
_UxCLabelTextSubclass1::_UxCLabelTextSubclass1(swidget
                 UxParent)
         :_UxCLabelText(UxParent,
                   InitialValue )
```

When you add an argument to the Interface Function of LabelTextSubclass1, you can use this argument to set properties in the Property Editor. So if you add an argument named anArg to the Subclass Interface Function, you can use it to set the anArg Core property:


Property Editor
File
Edit
View
Options
Help
Add Object:
Core
Initial Value
Source
Apply
This is the value passed to the LabelText constructor by the Subclass constructor.

LabelTextSubclass1 (LabelText)

	Initial Value	Source
AnArg	anArg	Private
Height	34	Private
Width	304	Private
X	814	Private
Y	298	Private

Figure 8-6 Constructor Arguments in the Property Editor

In the generated C++ code, the LabelTextSubclass constructor passes the anArg value to the LabelText constructor:

```
_UxCLabelTextSubclass1::_UxCLabelTextSubclass1( swidget
                                     UxParent, int anArg
                 )
:_UxCLabelText(UxParent, anArg)
```

Summary

The UIM/X Component Model allows you to build reusable components. From the creation of a top-level Component through the creation of Instances and Subclasses, the UIM/X Component Model provides a mechanism for building class hierarchies with inheritance, encapsulation, and polymorphism.

Working with Components, Subclasses, and Instances

9

Overview

Often one wants to reuse an interface element in several places. Simply duplicating the element to be reused would mean that subsequent changes would have to be repeated in many places.

A better approach is to build a Component as a separate interface and create an Instance of the Component in the other interfaces. In object-oriented terminology, the Component is a class, and an Instance an object of that class. It is also possible to create Subclasses of a Component, and hence create a class hierarchy using inheritance and method overloading.

In UIM/X, a Component is a top-level interface. You can create Instances of Components that are present either as interfaces in the project or as generated code. In the latter case, the code can be either compiled and linked to UIM/X or loaded into the Interpreter.

For example, consider a Bulletin Board interface that contains a Label and a TextField object. When an Instance of this Bulletin Board is used in another interface, the Instance looks and behaves as defined in the Component. Changes made to the Component are immediately reflected in all its Instances.

If you generate C++ code for the Component, it is a real C++ class. In C, the same behavior is achieved using the interface context mechanism.

Like classes in typical object-oriented languages, Components have a constructor, properties, and methods. The constructor is the function that builds Instances.

By default, an Instance has no editable properties—it inherits its properties from the Component. Properties that *are* editable in the Instance are made available by adding arguments to the Interface Function of the Component, or specifying property accessor methods. Any editable property of an Instance appears in the Property Editor in the Specific category.

Methods can be defined for Components, as they can be for any interface.

An Instance of any Component can be saved in the Palette for future use and shared among other users.

A Subclass of a Component is a top-level Instance of that Component. Like all top-level interfaces, a Subclass has its own constructor and properties.

Creating Instances of Components produces considerably less code than mere duplication. Instances work by calling the Interface Function of the Component they reference.

Note: Initial and final code written in the Declarations of a Component is executed when the Interface Function is called. UIM/X calls the Interface Function when you create, move, resize, or change the properties of an Instance of the Component. In general, the constructor is called every time you cause the Instance to be recreated.

By designating one object from the Component as a child site, you can make any Instance of the Component accept children.

Creating a Component

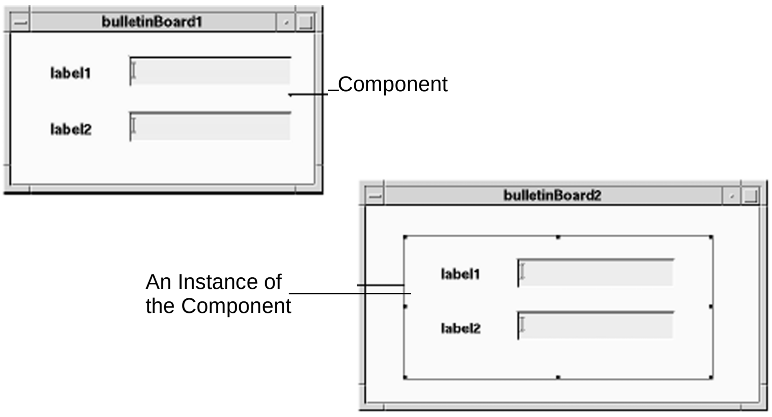


Figure 9-1 Components and Instances

Any top-level interface is a potential Component (see Figure 9-1). The following rules govern the use of interfaces as Components:

- A Component must take a swidget as the first argument of its Interface Function.
- The first argument to the Interface Function must be used as the Parent specified in the Declaration category of the Property Editor.
- The CreateManaged property of the Component should be set to `false`.

If set to `true`, the placement of the Instance in its parent can become unpredictable. Interfaces other than Components generally need this property set to `true`, so UIM/X automatically sets this property to `true` when an interface is created.

When you create an Instance, UIM/X automatically sets the CreateManaged property to `false` for the Component. This ensures that instances of the Component are placed properly.

Managing Instances

If you use an Instance or a Subclass as the start-up interface or as a top-level interface, you must use the method `VisualInterface_Manage()` to manage the interface.

The following example shows how to call the method:

```
swidget f1;
f1 = create_form1( NO_PARENT );
Interface_UxManage( f1, &UxEnv );
```

After you call `VisualInterface_Manage()`, you can use `UxPopdownInterface()` to hide the interface, and `UxPopupInterface()` to show it again. But you *must* call `VisualInterface_Manage()` to manage and display the interface for the first time.

Promoting an Object to Top-Level

If the object to serve as a Component already resides in an interface, you must first make it a top-level interface. Any object except a gadget or a menu can be promoted to a top-level interface.

To promote an object to a top-level interface, drag the object onto the desktop. A dialog will ask you if you want to replace the original object with an Instance of the Component.

Adding Editable Properties to an Instance

You add editable properties to an Instance by defining a pair of methods on the Component—one to set the property value and one to retrieve the property value.

These methods are known as *property accessors*. They comply with the Common Object Request Broker Architecture (CORBA), which specifies that properties be modeled as a pair of methods.

When you define a pair of property accessors, the property appears in the Specific category when you load an Instance into the Property Editor.

You can also add editable properties to an Instance by adding arguments to a Component’s Interface Function (constructor). These arguments are listed as the Core properties of the Instance in the Property Editor. If a property must be set before an Instance is created, add an argument to the Interface Function. Otherwise, use property accessors to add a property.

Note: Constructor arguments take precedence over accessor methods, so do not use the same name for a constructor argument as for an accessor method.

Defining Property Accessors

When you add an accessor property, you can store its value in an interface-specific variable, or use the accessor methods to encapsulate calls to `UxPut`*Property*`()` and `UxGet`*Property*`()` (or `XtVaSetValues()` and `XtVaGetValues()`).

For example, suppose you wanted to add an editable property called LabelString to the instances of a LabelText Component.

To Add a Property by Defining Accessor Methods

1. Open the Method Editor for the Component.
   a. Create the get accessor method:
   b. Choose Edit⇒Add Get Property.

      The Method Editor fills the Return Type, Name, and Code fields with appropriate default values.
   c. The default return type is `int`. Change this to `char *`.
   d. Enter the name of the property (in this case, LabelString) in the Name field.
   e. Enter the code required to retrieve the property value:

```
return UxGetLabelString( label );
```

   f. Click the Create Method button.

      The accessor method `_get_LabelString` is added to the Interface Methods list.
2. Create the set accessor method.
   a. Choose Edit⇒Add Set Property.

      Note that a default argument is declared for you in the Arguments section. (A property accessor always accepts a third argument, which is the property value.) Change the type of this argument to `char *`.
   b. Change the return type to `void`.
   c. Enter the code required to set the property value:

```
UxPutLabelString( label, value );
```

   d. Click the Create Method button.

      The accessor method `_set_LabelString` is added to the Interface Methods list.

Adding Interface Function Arguments

Suppose you have a Bulletin Board Component, and you want its AutoUnmanage property to be an Instance property. Because this property can be set only when the Bulletin Board is created, you add the property by adding an argument to the Interface Function.

Add a Property by Adding an Interface Function Argument

1. Select the Component and open its Declaration Editor.
2. Add an argument named `autoUnmanage` to the argument list in the Interface Function:

```
 swidget create_bulletinBoard1(swidget UxParent, char
                                         *autoUnmanage)
```

3. Load the Bulletin Board object into the Property Editor.
4. Enter:

```
      autoUnmanage ? autoUnmanage : "true"
```

as the value of the Core property AutoUnmanage and click OK.

Note: In the example above, a default value (true) for the AutoUnmanage property is specified. This ensures that if the value of `autoUnManage` is `NULL`, the value of the property is set to a valid value.

This technique avoids validation errors caused by setting properties to `NULL`. This kind of validation error occurs if you simply set the AutoUnmanage property to `autoUnmanage` since `NULL` is the default initial value for all Core properties of an Instance.

Note: Instances property values are matched to Component arguments by name. If you change an argument name, the values entered under the previous name are discarded.

Adding Pointer-to-Void Properties

Suppose you add a *pointer-to-void* property to a Component, either by adding an Interface Function argument or by defining property accessors.

When you enter a value for this `void *` property in the Property Editor, UIM/X passes the pointer directly to you (that is, to the Component's Interface Function or accessor method). It does not make a copy of the data. If you don't know whether or not the data pointed to will still be valid later, you should make your own copy of the data.

For example, if you enter a string in the Property Editor, you should use `UxCopyString()` in your Interface Function or accessor method to make a copy of this string.

Adding Callbacks to Instances

Callback accessors give you a way to install callbacks on the individual swidgets in an Instance. When you define a method named `Add`*EventName*`Proc()` on a Component, its Instances (and Subclasses) are given a Behavior property named *EventName*. Like a swidget callback, this property is set using the Callback Editor.

The job of a callback accessor is to install the callback defined in the Callback Editor. UIM/X passes the callback function and the client data to the method, so a callback accessor *must* accept two arguments:

- The callback function (an `XtCallbackProc` pointer).
- The callback client data (an `XtPointer`). UIM/X uses it to give the user access to the interface-specific variables and swidgets in the interface.

When you define a callback accessor, you must use the types `XtCallbackProc` and `XtPointer` for the arguments:

```
  XtCallbackProc cb;
  XtPointer client_data;
```

In the body of the method, you use `UxAddCallback()` (or `XtAddCallback()`) to register the callback:

```
  UxAddCallback( editButton, XmNactivateCallback, cb,
                                        client_data );
```

If you intend to change the callback function while the application is running, you may also want to add a call to `XtRemoveCallback()`. To do this, you will need to store the callback function and its client data in interface-specific variables. Each time the callback accessor is invoked, you can retrieve the previous values, pass them to `XtRemoveCallback()`, and then store the new values.

Setting Instance Geometry

In UIM/X, every Component is a Subclass of the `UxVisualInterface` class. The `UxVisualInterface` class is an abstract class defined by UIM/X. (See *"The Base Interface Class"* on page 107 for more information on the `UxVisualInterface` class.)

The purpose of the `UxVisualInterface` class is to give UIM/X and generated code a simple way to handle Instance geometry. It provides property accessors for the `X`, `Y`, `Height`, and `Width` properties. It also provides an interface method (`VisualInterface_Manage()`) for managing Instances.

If you need to change the size and position of an Instance (for example, in a callback), you must use these accessor methods. To invoke these inherited accessors, you use the method invocation macros defined for the `UxVisualInterface` class. For example, you set the x property of an Instance using the macro `VisualInterface__set_x()`:

```
VisualInterface__set_x( bulletinBoard1Instance1,
&UxEnv,920);
```

The method macros for the `UxVisualInterface` class methods are defined in *`uimx_directory`*`/include/UxInterf.h`.

Note: In generated C++ code, the `UxVisualInterface` class is a real C++ class. All C++ interface classes are derived from the `UxVisualInterface` class:

```
class _UxCbulletinBoard1: public UxVisualInterface
{
 // Class definition.
}
```

In UIM/X and in generated C code, the `UxVisualInterface` class is not represented by any data structure, but rather by a set of methods.

Creating an Instance

The following rules govern the creation of Instances as children of other objects:

- An Instance must never be placed in an object hierarchy where one of its ancestors is the Component. This circular nesting can send the Interpreter into an infinite loop.
- An Instance, like any other object, cannot be placed in an object that does not accept children. For example, Push Button objects do not accept children.

To Create a Child Instance

1. Select a Component.
2. Choose Selected Objects⇒Instance.

   OR

3. Choose Create⇒Instance from the Browser.

   The pointer becomes a left corner.

4. Drag the pointer over the parent to create the Instance.

   An Instance called `instance1` is created.

Instances of top-level Components can also be made, allowing for example a primary interface to “own” its own dialog box. See *“Calling Methods in Other Interfaces”* on page 144.

Creating a Subclass

A Subclass is a stand-alone interface that inherits all the content and methods of its base Component. Subclasses are used to build a class hierarchy where each Subclass can add its own children, properties, and methods, as well as inherit or override the methods it inherits from its base class.

To Create a Subclass

1. Select a Component.
2. Choose Create⇒Subclass of *interfaceName*.

The pointer becomes a left corner.

Note: When an interface that is usable as a component is selected, UIM/X offers to create an instance of that component. For example, if `drawingArea1` is selected, the Project Window Create menu displays Subclass of `drawingArea1` rather than Subclass.

---

3. Drag and draw the Subclass on the desktop.

   An icon labelled *interfaceName*Subclass1 appears in the Interfaces Area.

Creating Children of an Instance

You can add children to an Instance if its Component defines a child site. Within a Component (or Subclass), one object can be designated as a child site for other objects. To do this, you create a method called `childSite` which returns the designated child site object.

For example, consider `bb1`, a Component that consists of the hierarchy shown in Figure 9-2.

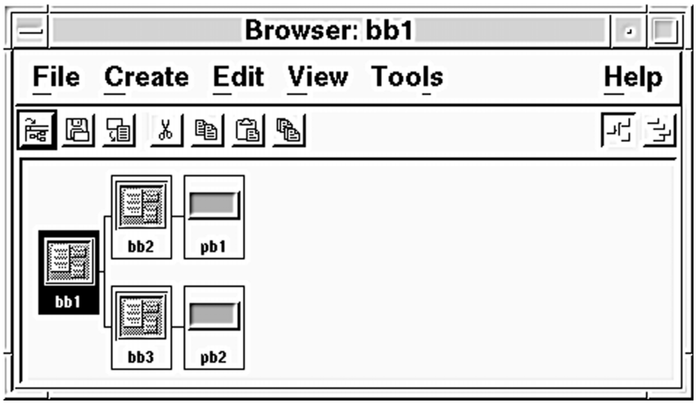
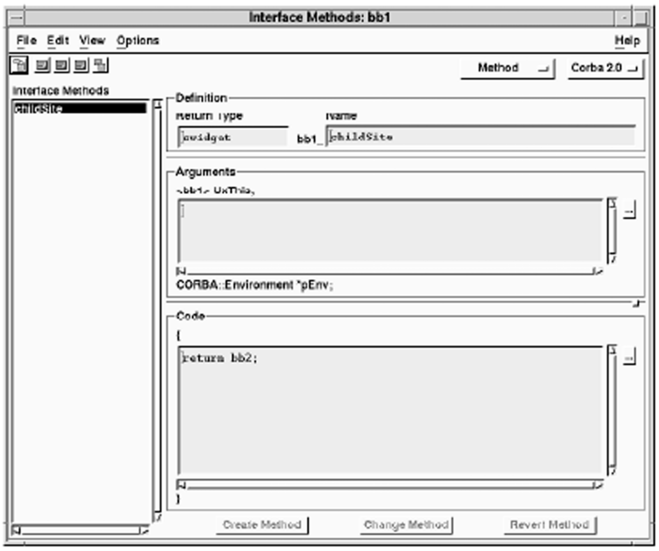


Figure 9-2 Adding Children to an Instance

If you want `bb2` to be the child site, open the Method Editor for `bb1`, and create a method called `childSite` as shown in Figure 9-3:



Interface Methods: childSite

Return Type: swidget bb1_ Name: childSite

```
<bb1> UxThis,

CORBA::Environment *pEnv;
```

```
{
return bb2;
}
```

Figure 9-3 ChildSite Method

If you create an Instance or Subclass of `bb1`, you can add children to this Instance.

Generating Code for Components

When you generate code for Components, Subclasses, and Instances, you must also generate an include file (that is, you must select the Generate Include File option on the Code Generation Options dialog box).

The include file is required because (in generated C code) the context structure of the Component is a part of the context structure of a Subclass. In generated C++, the context structure of the Component becomes the C++ base class for the Subclass.

For Instances, the Component's include file provides access to the Component's methods.

When the Component Exists Only as Code

A Component need not be directly available as a top-level interface loaded in the project. Components can exist as generated C or C++ code compiled and linked to UIM/X, or loaded into the Interpreter.

The following steps create a Subclass or an Instance whose Component interface exists only as code.

To Create an Instance of a Component that Exists as Code

1. Ensure that no interfaces are selected. Create an Instance by selecting Instance from the Selected Objects popup menu.

   Alternatively, create a Subclass by selecting Subclass from the Project Window's Create menu.

   Note that, contrary to the previous examples, there is no interface name appended to the Subclass and Instance menu entries. When using C code components you must first create a placeholder for the Instance or Subclass. Then, using the Property Editor, fill in the information required for the Instance or Subclass to compose a call to its Component's create function.

2. Place the Instance or the Subclass in the Property Editor.
3. Select Declaration from the Category menu.
4. Enter the name of the Component in the Component property.
5. Enter the Component's interface arguments in the `ArgDefinition` property.
6. If the Instance or Subclass is to have children, and the Component has a `childSite` method, enter the class name of the object designated as a child site in the `ChildSiteClass` property.
7. Enter the name of the Component's header file in the HeaderFile property.
8. Enter the Component's Interface Function in the Constructor property.

Note: In order for the Interpreter to locate the Interface Function it must be aware of the function's linkage. If the function's linkage differs from the Interpreter's mode, it must first be declared manually. See *"Registering Functions"* on page 171.

---

9. Press Apply.

Note: The `ArgDefinition` property must specify exactly the arguments to the Component's Interface Function, in their proper order.

Declaration Properties

The Declaration properties of an Instance or Subclass supply the information needed to compose a call to the constructor function of the component.

ArgDefinition	The `ArgDefinition` property specifies exactly the arguments specified in the Interface Function of the Component. These arguments are passed to the function that creates the Instance or Subclass of the Component. UIM/X keeps this property synchronized with a Component that is loaded in the project. If the Component exists only as C code, the arguments must be entered in the `ArgDefinition` property. The syntax of the `ArgDefinition` property is a sequence of standard variable declarations, entered as a string.
ChildSiteClass	The `ChildSiteClass` property specifies the class of the object designated as the Component's child site. See *"Creating Children of an Instance"* on page 132. For example, if the child site is a Form object, the `ChildSiteClass` property is set to `"form"`.
Component	The `Component` property specifies the name of the Component. If the Component interface exists, entering its name will automatically generate the `ArgDefinition` and `Constructor` property values appropriate for the Component.
Constructor	The `Constructor` property specifies the name of the function that creates instances of the Component (the Interface Function). The name is supplied automatically when the Component interface exists and its name is entered in the `Component` property of the Instance. If the Component exists only as code, the function must be entered—as a string—in the `Constructor` property.
CreateManaged	As for other objects.

HeaderFile The `HeaderFile` property specifies the name of the Component's generated header file. Generated code for the Instance or Subclass contains a reference to this header file. If no header file is specified the Component name is used.

Name The Instance or Subclass name.

Parent As for other objects.

PropDefinition The `PropDefinition` property is a list of declarations for the properties and callbacks of the Instance. It includes the `X`, `Y`, `Width`, and `Height` properties, as well as any properties and callbacks added by defining accessor methods on the Component.

Storing an Instance in a Palette

Instances can be stored in a Palette, and later created from the Palette, like any other object. To store an Instance in the Palette, you can either drag the Instance to a Palette or cut and paste from the clipboard.

You can also store Components and Subclasses in a Palette. However, saving a Component or Subclass (rather than an Instance) in the Palette merely duplicates the original, defeating the purpose of Components.

Note: If you want to use the Instances or Subclasses stored in a Palette, the Components used by the Instances and Subclasses must be available in UIM/X. The Component can be compiled into UIM/X, loaded as an interface file, or loaded as source code into the Interpreter.

Sharing Components Among Projects and Developers

Components (including Instances and Subclasses) can be shared among developers working on the same or related projects. They are three ways developers can share these building blocks:

- Distributing interface files.
- Distributing source files.
- Distributing object code and a Palette file.

Note: When you distribute Instances and Subclasses, you must also distribute the Components used by the distributed Instances and Subclasses.

Each method has advantages and drawbacks, discussed below.

Distributing Interface Files

The most direct method is to distribute the interface files containing Components. This method offers the following advantages:

- No effort is required of the Component creator.
- Each user can choose to load only the interface files required for his or her project.

The drawbacks of distributing interface files are the following:

- The interface files must be loaded each time a new UIM/X session is started.
- Interfaces from interface files require more memory than code.
- Users can edit the Components.

Distributing Source Files and a Palette File

Distributing the generated source files is a convenient way of sharing Components when there are many interfaces to be shared. The generated code is read into the Interpreter. Sharing source files has the following advantages:

- Source code is more flexible than compiled code; you can re-load modified source code during the same UIM/X session.
- The interpreted files do not require as much memory as interface files, and Instances perform better.

A drawback of source files is:

- Users must load the files into the Interpreter each time a new session is started.

To incorporate source files with their project, developers receiving the files must add the source file names to their project Makefile. This ensures that these source files are compiled when building the application. (This is done through the Program Layout Editor.)

The Palette file can contain one or more Instances that use the Component. Several Instances might be included if Specific properties differ.

Distributing Object Code and a Palette File

The surest method of distributing Components and Instances is to provide users with object (`.o`) files containing the Components and a Palette file of the Instances. The advantages of this method are:

- The difference between Motif widgets and Components will be transparent. Component creation parallels widget creation.
- Users will not be able to alter the Components.

The drawbacks are:

- Users must make an augmented UIM/X.
- Users must add the run-time object file to the project Makefile.

These two drawbacks can be overcome by making available a common augmented UIM/X executable and archive the run-time object file in `libux.a` for project linking.

Note: Two object (`.o`) files are required. The augmented UIM/X executable must be linked with an object file that is compiled with the `-DDESIGN_TIME` flag. (Do not compile the Components with the `-DUIMX_INTERNAL` flag.)

The object file that is archived in `libux.a` must *not* be compiled with the `-DDESIGN_TIME` flag. See Chapter 12, “Mixing Compiled and Interpreted Code.”

The Palette file can contain one or more Instances that use the Component. Several Instances might be included if Specific properties differ.

10 Methods

Overview

Methods provide the means by which application software can submit requests to an interface. To the C++ developer, the concept of methods is familiar.

Methods are different from regular functions in several ways. You can think of the interface as a C++ class where the methods are virtual member functions of the class. In fact, when you generate C++ code for your application, this is exactly what you get. Methods are polymorphic. This means that if you create a class hierarchy in UIM/X, a base class can define a method that can be either used or overwritten by subclasses of the base class. See *Chapter 9, "Working with Components, Subclasses, and Instances*." This is an important aspect of methods.

For example, suppose you wish to implement a Quit method for your application. If there are any unsaved changes in other interfaces belonging to the same application, the application is to prompt the user to save or discard them. To do this, create a base class for all your interfaces and define a Quit method in this base class. Subclasses of this base class requiring special handling when quitting can override the Quit method.

Methods can be called by actions, callbacks, other methods, in the Final Code section in the Declaration Editor, and from the Property Editor. You can mix methods from compiled and interpreted code. Methods can be defined for Components.

Methods can and should be used in place of auxiliary functions specified in the Declaration Editor in previous versions of UIM/X. Context pointer manipulation is handled automatically.

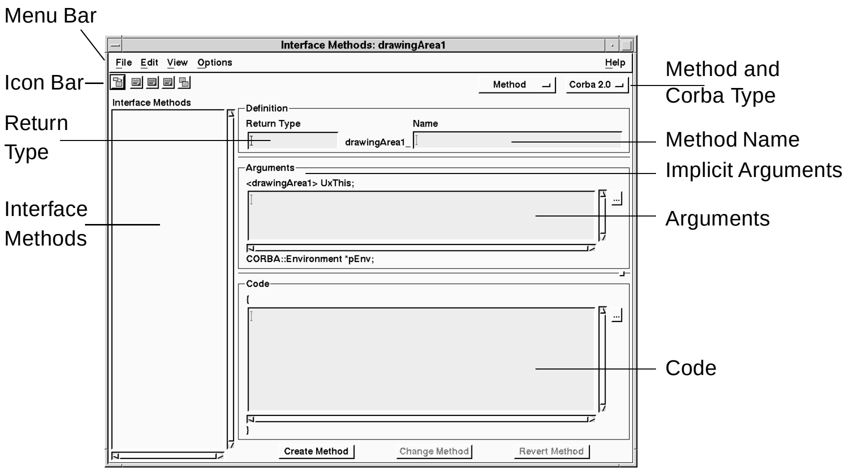
For developers familiar with object-oriented terminology: UIM/X provides support for dynamically dispatched methods, where each interface you build is a class that can include method definitions.

Keep the following language considerations in mind:

- If you use C, the generated code contains all the necessary support for invoking methods, just as it supports callbacks, translations, and actions.
- If you use C++, your methods are generated as virtual member functions of interface classes.
- If you use C but plan to migrate to C++, you can use methods now and simply switch to C++ code generation in the future.

Understanding the Method Editor

The Method Editor allows you to define and edit the methods of an interface. Figure 10-1 shows the main areas of the Method Editor.



```
<drawingArea1> UxThis;
CORBA::Environment *pEnv;
{
}
```

Figure 10-1 Method Editor

Area	Description
Menu Bar	Provides File, Edit, View, Options, and Help menus.
Icon Bar	Provides quick access to some of the most frequently used menu choices.
Interface Methods	Lists the methods defined for the selected interface. Selecting a method from the list displays the method's definition in the Method Editor's text fields.

Area	Description
Return Type	Shows the data type returned by the method. The signature of a method is formed from the Return Type, the Method Name, and the Arguments, if any arguments are specified. The Return Type is mandatory.
Method Name	Shows the method's name. Method names are of the form: *interface_name*, where *interface* is the name of the interface and *name* is supplied by the user. When you type a name into this field and press Return, the Method Editor verifies whether the method already exists. If the method exists, the Return Type and the Arguments fields are updated to show the signature of the method. You compose a new method by specifying a new name—the name does not appear in the Interface Methods list.
Implicit Arguments	Shows the arguments that all methods must accept.
Arguments	Shows a field in which arguments can be specified. The signature of a method is formed by the Return Type, the Method Name, and the Arguments, if any arguments are specified. Comments entered in this field are stripped when the method is parsed by the Interpreter. Enter your comments in the Code field instead.
Code	Shows a field in which the code for the method is specified. Within the method code, you can use all the swidget- and interface-specific variables defined in the interface.When you apply the method, this code is submitted to the Interpreter. UIM/X does not accept the method if syntax errors are detected.
Method Type	Specifies the type of method. The Property Set and Property Get types are property accessors. Property accessors are methods that set and retrieve the values of Instance properties. See *"Adding Editable Properties to an Instance"* on page 126 for more information on property accessors. All other methods use the Method type.
Corba Type	Specifies the type of Corba. The choices are Corba 1.1, Corba 2.0, and No Corba.

Creating, Changing, and Reverting Methods

The Create Method button creates a new method.

The Change Method button changes the current method (the method selected in the Interface Methods list). It applies all changes entered in the Method Editor to the current method.

Click the Revert Method button to cancel any changes that have not yet been applied. The Method Editor responds by displaying the method definition before any changes were initiated.

Reserved Words

You should not use the words `parent` or `name` anywhere in code specified in the Method Editor.

Overriding Methods in Subclasses

The methods you define in the Method Editor are *polymorphic*. In a class hierarchy, each Subclass can have a different version of the same method. When you redefine an inherited method and give it the same signature (the same arguments and return type), you *override* the inherited method.

To Override an Inherited Method

1. Load the Subclass into the Method Editor.
2. Type the name of the method you want to override in the Name field.
3. Press Return.

   The Method Editor fills in the Return Type and Arguments fields to match the signature of the inherited method.

   The Code area is blank—the method body is *not* copied.
4. Enter the code that defines this version of the method.
5. Click Create Method. The method is added to the list of methods for the Subclass.

When you override an inherited method, you can use the Implementors item on the Method Editor's View menu to display a list of classes that also implement the method. The popup menu on the Method List also displays the implementors. Note that these menus list only classes in the same class hierarchy.

Calling Methods

A method is called by its method macro, which when Corba 2.0 support (the default) is chosen, takes the form:

```
InterfaceName_MethodName(swidget, other_args, Environment *)
```

where:

InterfaceName_MethodName	The name of the method as assigned in the Method Editor
swidget	The first argument, always the top-level swidget of the interface to which the method is applied.
other_args	The user-defined arguments.
*Environment **	A pointer to an Environment structure (requirement for CORBA compliance). The type and location of the Environment structure depends on the CORBA support.

The Environment parameter is prescribed by the Common Object Request Broker Architecture (CORBA) bindings for C and C++. It can be used by a request broker to raise an exception when a method request fails. In UIM/X this parameter is present for source compatibility only; UIM/X never raises request broker exceptions. For convenience, a global Environment called `UxEnv` is defined in the header files `UxLib.h` and `UxDesign.h`. You can pass `&UxEnv` as the Environment parameter in all method calls.

Note: If you call a method not defined for an interface during design time, an appropriate error message is displayed. During run time, the call is ignored.

Calling Methods in Other Interfaces

In many applications, one interface needs to call methods defined by another interface. To do this, the interface that calls the methods needs access to both the method definitions and the top-level swidget of the interface that defines the methods.

Approach #1: Using Parented Top-Level Instances

The cleanest and most object-oriented way of allowing one interface to manipulate another is by adding an instance of the second interface to the first. This can be done in the same way that any instance is added to an interface; by selecting the interface to instantiate, choosing Selected Objects⇒Instance, and dragging the mouse pointer over the parent interface (see *“Creating Instances”* on page 95 for more information). If the interface being instantiated has an explicit shell or is a dialog, the instance will not be visible in the parent interface, but it will still be accessible through the Browser.

For example, suppose you have a main window that pops up an editor. Using the parented top-level instance approach, you would provide the editor with an explicit shell if it does not already have one, then you would add an instance of the editor to the main window. The `ActivateCallback` of the Push Button or menu item that pops up the editor can then do so with a single statement:

```
  VisualInterface_Manage( editorInstance1, &UxEnv );
```

or with a connection:

```
  ActivateCallback--->editorInstance1::Manage( &UxEnv )
```

You can then invoke any method on the editor instance using the name of the instance as the first argument:

```
  editor_Open( editorInstance1, other arguments, &UxEnv );
```

or by making a connection to the method:

```
ActivateCallback--->editorInstance1::Open(other arguments,
&UxEnv)
```

Approach #2: Using Manually-Created Instances

Alternatively, you can handle instance creation manually. This involves the following steps:

1. Insert code into the Includes, Defines, Global Variables area of the main window's Declaration Editor to declare the Interface Function and methods of the instance as appropriate. For example:

```
#ifndef DESIGN_TIME
#include "editor.h"
#else
extern swidget create_editor UXPROTO((swidget));
#endif
```

   In UIM/X, the methods defined by one interface are available to all other interfaces because the Interpreter stores all method definitions in the same translation unit. However, this is not true for the Interface Function, which must be declared as `extern`. In generated code, both an interface's methods and its Interface Function are declared in the interface's header file, and thus the file must be included in order to access these functions in generated code.

2. In the Interface Specific Variables area of the main window's Declaration Editor, declare a variable to hold the top-level swidget of the instance:

```
swidget myEditor;
```

3. In the Final Code section of the Main Window's Declaration Editor, make a call to the Interface Function of the instance, and store the result in the variable declared in the previous step:

```
myEditor = create_editor( NO_PARENT );
```

   See *"Styles of Handling Interfaces"* on page 77 for a discussion of alternate placements of the call to the Interface Function.

4. The `ActivateCallback` of the Push Button or menu item that pops up the editor can then do so with the single statement:

```
VisualInterface_Manage( myEditor, &UxEnv );
```

5. You can invoke any method on the editor instance using the name of the instance as the first argument:

```
editor_Open( myEditor, other arguments, &UxEnv );
```

   Note that connections cannot be used with this technique, as UIM/X does not recognize the instance.

Note: If the instance is created within the Interface Function of the main window as described here, the line

```
myEditor = create_editor( NO_PARENT );
```

must be interpreted in the Interpreter (with the selected interface set to the main window) to allow you to test the application in Test Mode. This is because UIM/X does not automatically call the Interface Function of the start-up interface in Test Mode, and so an instance of the editor must be manually created.

Approach #3: Creation During initialization

The approaches discussed above deal with the case where one interface creates a second interface and then invokes methods on the second interface. But suppose your application creates its interfaces during initialization (in the main program file). In this case, where do you store the top-level swidgets of the interfaces?

One solution is to store the top-level swidgets in variables declared in the main program. For example, if your application consisted of two interfaces, you could enter the following declarations in the Program Layout Editor:

```
/*-----------------------------------------------
 * Insert application global declarations here
 *-----------------------------------------------*/

#ifndef DESIGN_TIME

swidget Interface1;
swidget Interface2;

#include "interface1.h"
#include "interface2.h"

#else

swidget create_bulletinBoard1 UXPROTO((swidget));
swidget create_bulletinBoard2 UXPROTO((swidget));

#endif /* DESIGN_TIME */
```

When you create the two interfaces further down in the main program, you store the top-level swidgets of the two interfaces in the variables `Interface1` and `Interface2`:

```
   Interface1 = create_interface1( NO_PARENT );
   VisualInterface_Manage( Interface1, &UxEnv );

   Interface2 = create_interface2( NO_PARENT );
   VisualInterface_Manage( Interface2, &UxEnv );
```

This works at run time, but what about during design time? How can you invoke methods? The variables `Interface1` and `Interface2` won’t be available, and without these variables, you cannot invoke methods.

To make the variables `Interface1` and `Interface2` available in UIM/X, you need to load the following declarations into the reference translation unit of the Interpreter:

```
swidget Interface1;
swidget Interface2;
```

Declarations loaded into the reference translation unit are available to all interfaces. To learn how to load declarations into the reference translation unit, see *"Loading Files into the Reference Translation Unit"* on page 162.

Using Methods

The following example illustrates the use of methods.

A main interface, named MainPanel, opens two other interfaces by means of Push Buttons. These other interfaces are a command editor and a text editor. The main interface also has a Push Button labelled Quit to end execution. However, the main interface must first verify that the command editor and the text editor can be closed before the quit can be executed.

A method, called Quit, is defined for the main interface.

```
/* If either editor is open and refuses to    */
/* close, do not close the MainPanel window. */

if (CommandEditor && !CommandEditor_Quit(CommandEditor,
                                          &UxEnv))
{
     return 0;
}

if (TextEditor && !TextEditor_Quit(TextEditor, &UxEnv))
{
     return 0;
}
UxDestroyInterface(MainPanel);
return 1;
```

The Quit method is called from the `ActivateCallback` property of the Quit Push Button in the main interface.

```
MainPanel_Quit(MainPanel, &UxEnv);
```

The Quit method calls the Quit methods specified for the command editor and the text editor. The command editor, named CommandTool, has a Quit method that simply informs the main interface that it is quitting.

```
MainPanel_ChildHasQuit(mainParent, &UxEnv, CommandTool);
UxDestroyInterface(CommandTool);
return 1;
```

The text editor, named EditorTool, has a Quit method that first verifies whether text has been modified. If it has, a message is sent to inform the user that the changes must first be accepted. Then the text editor's Quit method informs the main interface that it is quitting.

```
/* Refuse to quit if the text was changed.   */

if (Modified)
{
     printf ("OK the text changes first.\n");
     return 0;
}

/* Tell main interface and then quit.        */

MainPanel_ChildHasQuit(mainParent, &UxEnv, EditorTool);
UxDestroyInterface(EditorTool);
return 1;
```

The variable `Modified` is declared in the Declaration Editor of the Text Editor and a callback in the Property Editor sets that variable when a change to the text occurs.

Both the command editor and the text editor call the ChildHasQuit method specified for the main interface. That method uses the argument `child` to reset both editors to `NULL`.

```
/* Message from child saying it is quitting. */
/* Note that the child is no longer up.      */

if (child == CommandEditor) CommandEditor = NULL;
else if (child == TextEditor) TextEditor = NULL;
```

Method Dispatch in Generated C Code

The C language does not directly support polymorphic method dispatch.

UIM/X provides this feature using several library functions used during interface construction to register the method implementations, and during method dispatch to invoke the appropriate implementation.

The generated C code for your interface registers the methods and defines the method macros that you use to call your methods.

This section outlines:

- Features inserted in the generated C code to support methods.
- The library functions that support methods.
- How to replace method dispatch software.

Features Inserted in Generated C Code

The following code fragments provide an overview of the method support that UIM/X inserts into the generated C files for interfaces that define methods.

In the header (`.h`) file:

Method identifier variables are declared, for example:

```
extern int UxMyIface_MyMeth_Id;
```

Method macros are defined if they are not already defined, for example:

```
#ifndef MyIface_MyMeth
#define MyIface_MyMeth(UxThis, Arg1, Arg2, pEnv) \
 ((Type(*)()) UxMethodLookup(UxThis,
                        UxMyIface_MyMeth_Id,
    MyIface_MyMeth_Name))(UxThis, Arg1, Arg2, pEnv)
#endif
```

In the source (.c) file:

Each method is generated as two static functions. One function contains the code entered by the user in the Code area; the other one handles context swapping and calls the first. For example:

```
static Type Ux_MyMeth(swidget UxThis, Arg1Type Arg1,
                     Arg2Type Arg2, Environment *pEnv)
{
/* method body including code from Method Editor */
}

static Type _MyIface_MyMeth(swidget UxThis, Arg1Type Arg1,
                     Arg2Type Arg2, Environment *pEnv)
{
     _UxCmyIface *UxSaveCtx = UxMyIfaceContext;

 UxMyIfaceContext = (_UxCmyIface *) UxGetContext( UxThis );
 if (pEnv)
         pEnv->_major = NO_EXCEPTION;
 Ux_MyMeth( UxThis, Arg1, Arg2, pEnv);
 UxMyIfaceContext = UxSaveCtx;
}
```

Each Instance of this class—that is, the class generated by this interface function—is assigned a class identifier:

```
static int _UxIfClassId;
```

The following statement is executed once:

```
_UxIfClassId = UxNewInterfaceClassId();
```

Note: For interfaces that are actually Subclasses, the class id registration is based on the baseClass class id. The code looks like:

```
_UxIfClassId = UxNewSubclassId( UxGetClassCode
(MyIfaceSubclass1));
```

The following statement is executed once for each top-level swidget built. ClassCode is merely a field in the swidget object:

```
UxPutClassCode( MyIface, _UxIfClassId );
```

The method identifier is initialized by registering the method body as the implementation of the named method for this class. This initialization occurs only once.

```
UxMyIface_MyMeth_Id = UxMethodRegister( _UxIfClassId,
     UxMyIface_MyMeth_Name, (void (*)()) _MyIface_MyMeth);
```

Library Functions Used to Support Methods

The library functions maintain three sets of data:

- A mapping of method names to unique integer identifiers (MethodIds).
- A two-dimensional method lookup table in which the function pointer for a given class' implementation of a given method is stored at the coordinates [ClassId, MethodId].
- A table which maintains the class hierarchy.

The library functions are `UxNewInterfaceClassId()`, `UxNewSubclassId()`, `UxMethodRegister()`, and `UxMethodLookup()`. They are described below.

UxNewInterface ClassId

```
int UxNewInterfaceClassId()
```

which returns a new class identifier for a subclass of VisualInterface.

UxNewSubclassId

```
   int UxNewSubclassId(int baseClassId)
```

which returns a unique integer each time it is called and creates the association between the base class and the new subclass id returned.

UxMethodRegister

```
   int UxMethodRegister(int ClassId, char* MethodName,
                   void (FunctionPtr*)())
```

which:

- Maintains the mapping of MethodNames to MethodIds either finding the name in the registry or adding a name and assigning it a new MethodId.
- Enters FunctionPtr in the method lookup matrix at the coordinates [ClassId, MethodId].
- Returns the MethodId.

UxMethodLookup

```
   UxMethodLookup(swidget sw, int MethodId, char
*MethodName)
```

which:

- Extracts the ClassCode from the given swidget.
- Returns the function pointer at the coordinates [ClassCode, MethodId] in the method lookup table.

For each method, UIM/X generates a macro of the form `Interface_Method` that uses `UxMethodLookup` to call the method. For example:

```
   #define MyIface_MyMeth( UxThis, pEnv ) \
        ((int(*)())UxMethodLookup(UxThis,
UxMyIface_MyMeth_Id, \
        UxMyIface_MyMeth_Name)) ( UxThis, pEnv )
```

Replacing Method Dispatch Software

You can install your own method dispatching scheme by replacing the functions listed above. Your substitute functions must satisfy the following requirement.

The result of executing

```
   int baseClassId = UxNewClassId();
   int cid = UxNewSubclassId(baseClassId);
   int mid = UxMethodRegister(baseClassId, "name", func);
   UxPutClasscode(sw, cid);
```

should be such that a subsequent call to

```
   UxMethodLookup(sw, mid, "name");
```

returns the same `func` function pointer passed to `UxMethodRegister()`.

Methods in C++ Code

When C++ code is generated, methods are generated as virtual methods of the interface class. (You can change how UIM/X generates member functions. See *"Generating Member Functions for Methods"* on page 184.)

For example, if you define the following method

```
int MyIface_Quit (swidget UxThis, char *msg, Environment
*pEnv);
```

the generated code has in its class definition a virtual method as follows:

```
   class _UxCMyIface : public UxVisualInterface
   {
   public
        // User Defined Methods
        virtual int Quit (char * msg, CORBA::Environment * pEnv);

             // Rest of class definition
   };
```

Note: Note that the first argument to the method is no longer present in the generated C++ code. UIM/X calls methods by passing the swidget of the top-level interface as the first argument to the method. This is how we support method dispatching in the C language. Because this is a built-in behavior in C++, a macro is generated to translate a method call on an interface swidget into a member function call on the interface object.

For example:

```
#ifndef MyIface_Quit
#define MyIface_Quit( UxThis, pEnv, msg ) \
(((_UxCmyIface *) UxGetContext(UxThis))->Quit(pEnv, msg))
#endif
```

11 Using the Interpreter

Overview

The built-in Interpreter allows you to test the behavior of an interface without having to suffer through a tedious compile and relink stage. By switching to Test mode, code entered anywhere can be tested immediately.

The Interpreter can be used directly. However, it is more common for the Interpreter to be used implicitly to evaluate code when testing an interface in Test mode. Typically the Interpreter Work Area is used to declare temporary variables, to change values for test purposes, and so on.

Translation Units

The Interpreter supports multiple, independent translation units for a project. A *translation unit* is analogous to a source file together with any included files. You can use conditional compilation to control the contents of a translation unit.

The Interpreter supports three types of translation units:

- An *interface translation unit* is associated with each interface.
- The *reference translation unit* is shared by all interfaces and editors in UIM/X.
- The *general translation unit* is a general work area used by the Interpreter.

.

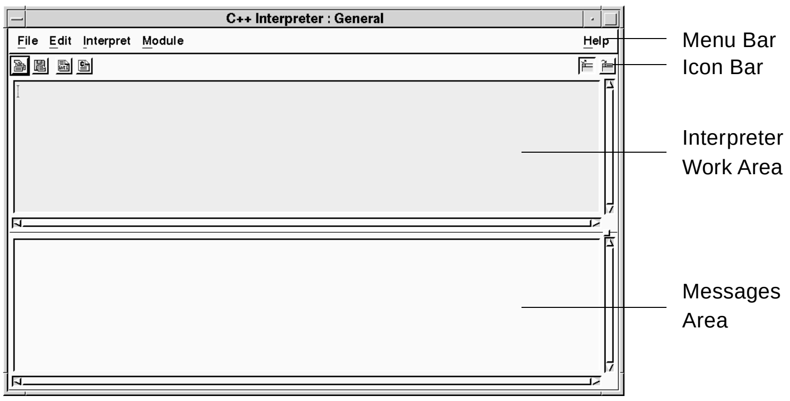


Figure 11-1 Interpreter

Interface Translation Unit

Each interface in a project has all its source code (object variable declarations, callbacks, and so on) gathered into an interface translation unit. Static functions and variables are not shared across interfaces. When a new interface is created, it is given its own translation unit.

To Work with a Translation Unit for a Specific Interface

1. Select the interface by clicking on it, or by clicking on its icon in the Interfaces area of the Project Window.
2. Click on the Selected Interface icon in the Interpreter's icon bar, or choose Module⇒Selected Interface from the Interpreter.

Reference Translation Unit

The reference translation unit contains the declarations and definitions common to all translation units. The reference translation unit is shared by all interfaces and editors in UIM/X.

UIM/X uses the reference translation unit to include the standard UIM/X, Motif, Xt, X, and system header files for use in all translation units.

You cannot use the Interpreter work area to declare and evaluate code in the reference translation unit. However, you can load definitions and declarations into the reference translation unit. See *"Loading Files into the Reference Translation Unit"* on page 162 for more information.

General Translation Unit

The general translation unit contains the declarations and definitions entered via the Interpreter Window when the General Module is selected. The general translation unit also includes all code entered in the Action Table Editor.

To Work with the General Translation Unit

1. Click on the General icon in the Interpreter's icon bar, or choose Module⇒General from the Interpreter.

Evaluating Code with the Interpreter

Whether you are making declarations or evaluating expressions, operations in the Interpreter are quite similar.

The Interpreter only evaluates or declares code that is highlighted in its work area. Code can be typed directly into the Interpreter Work Area or loaded from a file.

Declaring Code

Declaring code makes variables, typedefs, and function definitions known to the Interpreter.

To Declare Code

1. Enter the code in the Interpreter Work Area.
2. Highlight the code you have entered using the Select mouse button.
3. Click on the Declare icon in the Interpreter's icon bar, or choose Interpret⇒Declare from the Interpreter, or choose Interpreter Text⇒Declare.

Evaluating Expressions

To Evaluate an Expression

1. If the expression you want to evaluate contains variables local to a specific interface, select that interface and choose Module⇒Selected Interface in the Interpreter.
2. Enter the expression in the Interpreter Work Area.
3. Highlight the expression using the Select mouse button.
4. Click on the Evaluate icon in the Interpreter's icon bar, or choose Interpret⇒Evaluate from the Interpreter, or choose Interpreter Text⇒Evaluate.

   The result displays in the Messages Area.

Note: Because the Interpreter can evaluate only one expression at a time, enclose multiple statements in curly brackets ({ and }).

The symbol `UX_INTERPRETER` is automatically defined at UIM/X start-up. You can use it to conditionally execute code by the Interpreter. See *"Protecting Code from the Interpreter"* on page 166 for more information.

Viewing Interpreter Results

The results of selecting Declare or Evaluate from the Interpreter are displayed in the Interpreter Messages Area. For example, the value returned by a function is displayed in the Interpreter Messages Area. Any output to `stderr` or `stdout` is displayed in the Project Window Messages Area.

Clearing the Interpreter Work Area or the Messages Area

To clear the Interpreter Work Area, choose Edit⇒Clear Interpreter Text from the Interpreter, or choose Interpreter Text⇒Clear.

To clear the Interpreter Messages Area, choose Edit⇒Clear Messages from the Interpreter, or choose Messages⇒Clear.

Opening Files and Loading Source Code

The Interpreter allows you to:

- Open a file into the Interpreter Work Area without evaluating or declaring the code.
- Load files into the general translation unit.
- Load files into the reference translation unit.

Opening Files in the Interpreter Work Area

When you load a file using Open, you can edit the file, highlight portions of it, and declare or evaluate those portions.

To Open a File in the Interpreter

1. Click on the Open icon in the Interpreter’s icon bar, or choose File⇒Open from the Interpreter.

   A File Selection box appears.

2. Select a directory and a file name and click on OK.

Loading Source Code into the General Translation Unit

When you load source code into the general translation unit, the source code is declared automatically. The contents of the file are not displayed in the Interpreter Work Area.

To Load a File into the Interpreter

1. Choose General from the Interpreter Module menu.
2. Choose File⇒Load Source Code from the Interpreter.

   A File Selection box appears.

3. Select a directory and a file name and click on OK.

Loading Files into the Reference Translation Unit

Loading header files into the reference translation unit makes definitions and declarations available to all interfaces and editors.

To Load Header Files Using a UIM/X Resource

The UIM/X resource `GlobalIncludes` specifies the header files loaded into the reference module at start up. This resource specification is already in the UIM/X resource file. You just have to uncomment it by removing the exclamation point (!).

The resource `GlobalIncludes` specifies a comma-separated list of header files:

```
Uimx3_0.GlobalIncludes:          mydefs1.h,mydefs2.h
```

It will accept full or relative path names. If no path name is specified, UIM/X uses the Interpreter include paths to search for the files. (The Interpreter include paths can be specified using the `-I` flag in the UIM/X `cflags` resource.)

To Load a Header File Manually

1. Set the UIM/X resource `UxInterpSharedDefinitions.set` to `false`:

```
Uimx3_0*UxInterpSharedDefinitions.set: false
```

This resource setting adds the Load Shared Definitions menu item to the Interpreter’s File menu.

Note: See the *UIM/X Installation Guide* for more information on application defaults and the UIM/X resource file.

2. Start UIM/X and open the Interpreter window. *Do not load your project, interfaces, or palettes first*.
3. Select File⇒Load Shared Definitions from the Interpreter.

   A file selection box appears.

4. Use the file selection box to select the header file and click OK.

   The Interpreter loads the header file into the reference module.

The disadvantage of this approach is that you have to load the header files every time you start or reset UIM/X.

Calling Functions with the Interpreter

During Test Mode, action and callback code in your interfaces can be executed by the Interpreter. This interpreted code can call compiled functions, provided those functions are found in the UIM/X executable. Many of the standard X, Xt, Xm and system functions are used by UIM/X and are a part of the executable.

However, you may want to call a standard function or a function in the compiled application that is not already included. To make this possible, you can make an augmented UIM/X executable as described in Chapter 12, “Mixing Compiled and Interpreted Code.”

When the Interpreter encounters a function that is unknown to it, it searches for the function in the executable. This search may take several seconds the first time that function is encountered. Subsequent calls will be at normal speed.

To avoid the search of the executable, make external functions known to the Interpreter using `UxRegisterFunction()` in your *uimx_directory*`/config/uimx_main.cc` file. Similarly, `UxRegisterGlobal()` makes external variables known to the Interpreter. The commented examples at the end of `uimx_main.cc` illustrate their use.

Another useful Ux Convenience Library function is `UxAddIncludePath()`, which adds a new path to those searched by the Interpreter for include files.

The Interpreter and Run-time Errors

The built-in Interpreter automatically recovers from run-time error conditions in interpreted code such as segmentation violation, bus errors, and floating point errors. It does this by catching the signal, freeing up the Interpreter stack, and returning to the point where the Interpreter was first called. This recovery mechanism also works when an error occurs in a compiled function that was called from the interpreted code.

You can, however, generate conditions from which it is not possible to recover. When this happens, UIM/X attempts to save your interfaces in a file written to the `/tmp` directory and then exits.

The following are some of the conditions where UIM/X may fail to recover.

- UIM/X's own data structures, as well as those of a linked-in application, may be corrupted by assignment through invalid pointers (whether from compiled or interpreted code). This may eventually cause UIM/X to exit.
- Calling X or UIM/X functions with incorrect arguments can lead to the creation of invalid data structures that will cause error conditions later in code not called through the Interpreter. Such errors cannot be handled by the Interpreter's recovery mechanism.
- Errors in program code called from compiled interface components are not recoverable.
- When an interpreted function is called (through a pointer) from compiled code, error conditions in the interpreted code—or in subsequently called compiled code—are not recoverable, unless the compiled caller was in turn called from the interpreted code. This condition can occur in an interpreted input handler for subprocess control.

Note: If a call is made that would cause an ordinary program to crash, UIM/X will also crash because the Interpreter doesn't have control over code that is compiled into UIM/X.

The Interpreter Mode and Code Generation

The Interpreter mode (ANSI, K&R, or C++) must be compatible with the language selected on the Code Generation Options dialog box. For example, if you intend to generate ANSI C code, you should set the Interpreter to ANSI C mode before you begin developing your interfaces. For best results, choose the Interpreter mode that matches the code generation option.

If you create and save interfaces in ANSI mode, and then switch the Interpreter to K&R mode, you will have to update all Interface Function declarations. This is because generated Interface Functions will have an ANSI-style parameter list:

```
swidget create_drawingArea1(swidget UxParent);
```

When you develop interfaces in ANSI or C++ mode, the generated `create` (or `popup`) interface functions have ANSI-style declarations. If you switch the Interpreter mode to K&R and load the interfaces, UIM/X reports an error each time it encounters an ANSI-style declaration.

Notes about the Interpreter

The following should be noted about the Interpreter:

- The Interpreter Work Area is like an infinite length file—a file that the compiler never stops reading. As a result, the Interpreter Work Area doesn’t allow operations which would not be allowed by a compiler in a single file—except evaluating expressions, of course.
- After you declare a symbol in the Interpreter Work Area, you cannot change the declaration. So, for example, after you declare an array with a fixed size, you cannot change the size of the array.
- After you introduce a symbol into the Work Area, you cannot remove the symbol. The only way to clear the translation units is to reset UIM/X.
- When you create a new interface, you can test its Interface Function by typing it into the Interpreter Work Area and evaluating it. The interface must be selected in the Interfaces area of the Project Window and by choosing Module⇒Selected Interface from the Interpreter. This is useful for testing, especially when arguments are being passed to the Interface Function, or when you have entered initial or final code. Calling an interface function in a callback is another way to evaluate the initial and final code. The code is also executed each time you create an instance of a Component.
- The object name is used as a variable name by the Interpreter to refer to the object. Each object name must therefore be a valid variable name.
- To reference objects in another interface, first make them global.
- Changes made with the Interpreter do not affect the values established in the Property Editor, which are initial values.

- You can hang UIM/X by putting the Interpreter into an infinite loop. The following is a simple example:

```
      while (1)
      {
          printf("Why?\n");
      }
```

  If you kill the process to exit this loop, you lose any unsaved changes. To avoid losing any of your changes, you can send a segmentation violation signal (signal 11). You do this by executing the following UNIX command:

```
      kill -11 process_number
```

  In some cases, the Interpreter will get the signal and break from the infinite loop. When this happens, UIM/X attempts to save your interfaces in a file written to the `/tmp` directory and then exits.

Using CFLAGS

Flags for the Interpreter can contain environment variables. The `-D` flag can be specified to define a symbol. The `-D_NO_PROTO` flag suppresses prototypes in the standard library. The `-I` flag can be specified to tell the Interpreter where to search for include files.

Protecting Code from the Interpreter

At times you may wish to conditionally evaluate code with the Interpreter. For example, suppose you wish to include a header file that is not yet complete. To allow you to use the Interpreter to conditionally evaluate code, UIM/X automatically defines the symbol `UX_INTERPRETER`. This symbol is useful, but you must be careful when using it.

To insert an `#ifndef UX_INTERPRETER` protection block in your code:

1. In any UIM/X Text Editor, place the text cursor at the spot where you want to insert the protection block.
2. Press Control-Z.

UIM/X inserts the following lines:

```
   #ifndef UX_INTERPRETER
   #endif /* UX_INTERPRETER */
```

To surround an existing block of text with `#ifndef UX_INTERPRETER` protection:

1. Select the block of text you want to surround.
2. Press Control-W.

   UIM/X surrounds the selected text with `#ifndef UX_INTERPRETER` protection.

The `UX_INTERPRETER` protection block is automatically stripped by UIM/X during code generation. In order for it to be found and stripped, the block must be inserted properly every time. Therefore, to effectively use the `UX_INTERPRETER` protection block, you must abide by the following rules:

- Use the Control-Z and Control-W accelerators. Do not type in the `UX_INTERPRETER` protection block manually.
- Do not edit the `#ifndef` or `#endif` lines of an `#ifndef UX_INTERPRETER` protection block.
- Do not add an `else` clause to an `#ifndef UX_INTERPRETER` protection block.

12 Mixing Compiled and Interpreted Code

Overview

The UIM/X executable can be augmented with the object code of other applications. In particular, you can compile code generated by UIM/X and link it into the UIM/X executable.

Linking object code with UIM/X gives the Interpreter access to the functions in the object code. The Interpreter can execute any compiled function (or method) contained within the UIM/X executable.

Augmenting UIM/X

Augmenting UIM/X allows you to:

- Simplify the development of an interface for an application program. You can design the application's interface, insert calls to the compiled application functions, and test the interface, all without having to exit UIM/X.
- Link in Components distributed in object form.
- Link interfaces into the development environment.

  Large projects have many interfaces. As individual interfaces are finished, you can remove them from the project and make them part of the UIM/X development environment. As the project progresses, there will be fewer interfaces to load, edit, and test.

  To do this, you generate the code for the interface, compile it, link it with UIM/X, and remove the interface from the project (keep a backup copy of the interface's `.i` file).

  Note – If you generate C++ code, you need to generate Ux Integration code. See *UIM/X Advanced Topics* for more information.

  UIM/X allows you to mix compiled and interpreted code, so you can still test the entire project—the interfaces you load and create interactively can call the create functions of the interfaces that exist only as object code.

- Use a compiled interface as an editor within UIM/X. Suppose you use UIM/X to create a specialized widget editor. You can make this editor a part of UIM/X by compiling its generated code and linking it with UIM/X.

The makefile template */usr/$UIMXDIR*/`config/Makefile.uimx` allows you to augment UIM/X with C and C++ object files and libraries.

Note – *$UIMXDIR* refers to the folder where UIM/X is installed.

Note – The object code should not contain a `main()` function. Any initialization required by the application can be done from within the `main()` function in */usr/$UIMXDIR*/`config/uimx_main.cc`.

If you need to access the internal data structures of the swidget classes in an augmented UIM/X, you must make sure that the symbol `PRIVATE_SWIDGET` is defined when you compile a UIM/X executable. You can do this by adding the flag `-DPRIVATE_SWIDGET` to the `cflags` resource or to one of the makefile macros in `Makefile.uimx`.

Registering Functions

The file `/usr/$UIMXDIR/config/uimx_main.cc` contains the function `UxRegisterFunctions()`. You register a function with the Interpreter by inserting a call to `UxRegisterFunction()` in `UxRegisterFunctions()`.

Registering a function has two advantages:

- It makes the address of the function known to the Interpreter, eliminating the delay associated with looking up the function the first time it is encountered.
- It ensures that functions from other libraries are included in the executable, and are thus accessible from the Interpreter. This is important when you link another library with UIM/X. If you do not reference the symbols in the library, some modules may not get linked into the UIM/X executable.

`UxRegisterFunction()` is declared as follows:

```
void UxRegisterFunction(char *name, void
(*fptr)());
```

The parameter `name` is the name of the function, and `fptr` is a pointer to the function.

When you register a function, you must also declare it. You can do this by including the appropriate header file in `uimx_main.cc`, or adding an `extern` declaration after the declaration of `UxRegisterFunction()`. The following example illustrates both approaches:

```
#include <math.h>

    void UxRegisterFunction();
    extern char *MarksVryXcllntFn(void);

    void UxRegisterFunctions()
    {

        UxRegisterFunction("sin", sin);
        UxRegisterFunction("MarksVryXcllntFn",

MarksVryXcllntFn);
    }
```

To use a registered function in UIM/X, you must declare the function before calling it. You can either explicitly declare the function, or include a header file that contains the required declaration.

Note – Ensure that the function you are preregistering has been declared with its proper linkage. A C function must be declared as `extern "C"`.

Registering Globals

The file */usr/$UIMXDIR*/`config/uimx_main.cc` contains the function `UxRegisterGlobals()`. You register a global with the Interpreter by inserting a call to `UxRegisterGlobal()` in `UxRegisterGlobals())`.

Registering globals has the same advantages as registering functions.

`UxRegisterGlobal()` is declared as follows:

```
void UxRegisterGlobal(char *name, char *gptr);
```

The parameter name is the name of the variable, and gptr is a pointer to the variable.

When you register a global, you must also declare it. You can do this by including the appropriate header file in uimx_main.c, or adding an extern declaration after the declaration of UxRegisterGlobal(). The following example illustrates the second approach:

```
void UxRegisterGlobal();
extern int MarksGlobal;

void UxRegisterGlobals()
    {
    UxRegisterGlobal("MarksGlobal", &MarksGlobal);
    }
```

Note – Ensure that the global you are preregistering has been declared with its proper linkage. A C global must be declared as extern "C".

Conditional Compilation in Generated Code

When you compile generated code and link it with UIM/X, you may want to avoid certain function calls. A good example is XtCloseDisplay(). Calling this function during testing will terminate the UIM/X session. You can use the DESIGN_TIME symbol to control compilation:

```
#ifndef DESIGN_TIME
XtCloseDisplay();
#endif
```

When you use Makefile.uimx to augment UIM/X (see below), this symbol is defined.

Using Makefile.uimx

If you examine */usr/$UIMXDIR/*config/Makefile.uimx, you will see that the makefile contains a limited number of macro variables. The rules and additional macro variables required to build an augmented UIM/X are contained in the makefile *`/usr/$UIMXDIR`*`/mkinclude/central.mk`, which is included at the end of `Makefile.uimx`.

The macros in */usr/$UIMXDIR*`/config/Makefile.uimx` define the target and dependent files for augmenting the UIM/X executable. The following table describes these macros.

Macro Name	**Definition**
`AUGEXEC`	The name of the augmented UIM/X executable. In *`/usr/$UIMXDIR`*`/mkinclude/central.mk`, `$(AUGEXEC)` is the target that builds an augmented UIM/X.
`AUGMAIN`	The object file for the main program file of the augmented executable.
`APPL_OBJS`	A list of C object files to be linked with UIM/X.
APPL_CPLUSOBJS	A list of C++ object files to be linked with UIM/X.
`EXTRA_CFLAGS`	Use this macro to define extra C compiler options required for compiling the files `$(APPL_OBJS)`. By default, this macro sets the `-DDESIGN_TIME` flag. Generated code must be compiled with the `-DUIMX_INTERNAL` flag to make an interface into an editor in UIM/X. You can also use this macro to add the `-DPRIVATE_SWIDGET` flag.
`EXTRA_CPLUSFLAGS`	Use this macro to add C++ compiler flags.
`EXTRA_LDFLAGS`	Use this macro to define any extra link editor options required for linking object code with UIM/X.
EXTRA_UXLIBS	Use this macro to list the libraries you want linked into the UIM/X executable.

General Procedure for Using Makefile.uimx

The general procedure for using the makefile *a/usr/$UIMXDIR*`/config/Makefile.uimx` is as follows:

1. Create a working directory.
2. Copy */usr/$UIMXDIR*`/config/Makefile.uimx` to the file `Makefile` in your working directory. Renaming the makefile allows you to invoke `make` without specifying the name of the makefile.
3. Copy the file */usr/$UIMXDIR*`/config/uimx_main.cc` to your working directory. Insert any required initialization code in the file. The comments in the file indicate where such code should be inserted.
4. Copy the source (or object) files you want to compile and link with UIM/X to your working directory.
5. Modify the makefile macros described in the above table. Use the macros to name the executable and to list the object file for each source file in your working directory.
6. Use `touch` to ensure that all dependent files are more recent than the target.
7. Invoke `make`. Use the value of the macro `AUGEXEC` to specify the target.

Using central.mk

The makefile */usr/$UIMXDIR*`/mkinclude/central.mk` contains the rules and additional macro definitions required to augment UIM/X. This makefile is included by `Makefile.uimx`.

The target and rule lines in `central.mk` that build an augmented executable are shown below for C++:

```
AUGMAINOBJ = $(AUGMAIN:.cc=.o)

$(AUGMAINOBJ): $(AUGMAIN)
        $(CPLUS) -c $(CPLUSFLAGS) $(AUGMAIN)

$(AUGEXEC): $(APPL_OBJS) $(APPL_CPLUSOBJS)
$(UIMXOBJ)
    $(CPLUS) $(LDFLAGS) $(EXTRA_LDFLAGS) -o $@ \
    $(APPL_OBJS) $(APPL_CPLUSOBJS) $(UIMXOBJ)
$(LIBS1)
```

and for C:

```
AUGMAINOBJ = $(AUGMAIN:.c=.o)

$(AUGMAINOBJ): $(AUGMAIN)
    $(CC) -c $(CFLAGS) $(AUGMAIN)

$(AUGEXEC): $(APPL_OBJS) $(UIMXOBJ)
    $(CC) $(LDFLAGS) $(EXTRA_LDFLAGS) -o $@ \
$(APPL_OBJS) $(UIMXOBJ) $(LIBS1)
```

The macros `AUGEXEC`, `AUGMAIN`, and `APPL_OBJS` are defined in *usr/$UIMXDIR*`/config/Makefile.uimx`.

The other macros in the above lines are defined in `central.mk`. You can edit some of the macros to tailor the compilation and linkage of an augmented executable.

Advanced C++ Programming in UIM/X

A

Overview

This chapter describes the issues associated with using UIM/X for C++ development. It starts by explaining how to configure UIM/X for C++ development. It discusses the Class View in the Declaration Editor, and how to generate member functions for methods. Lastly, it outlines some of the restrictions of programming with C++ in UIM/X.

Configuring UIM/X for C++ Development

UIM/X provides a set of features to make it easier to program with C++. By default, these features are disabled. To use these features, you must uncomment several resource specifications in the UIM/X resource file *uimx_directory*`/app-defaults/Uimx3_0`. The following table describes these resources.

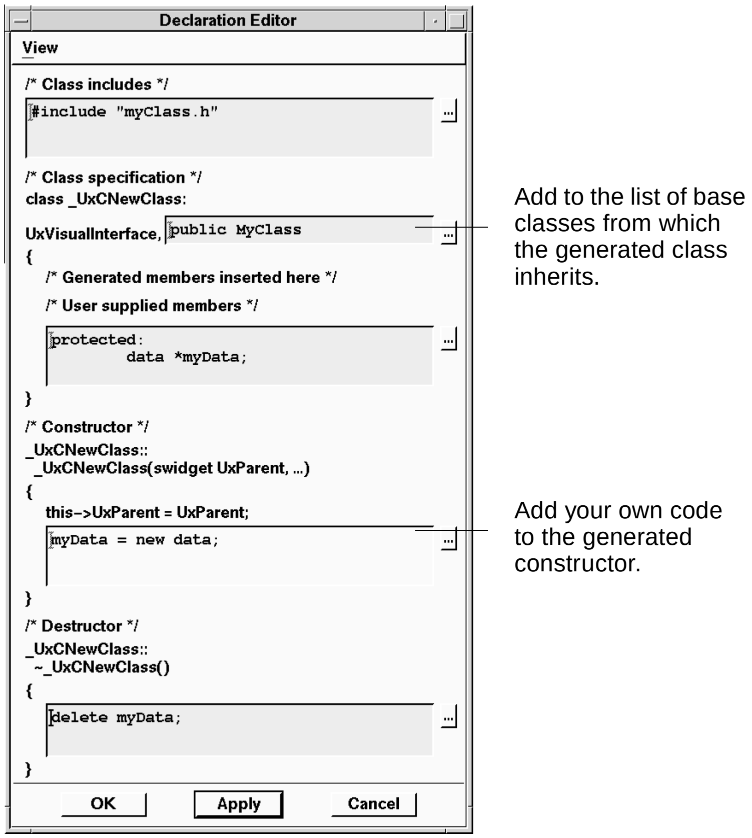
Resource	**Feature**	**For More Information...**
`UxMEMethodSpec.set`	Control whether a method is virtual or nonvirtual.	*"Generating Member Functions for Methods"* on page 184
`UxMEAccessSpec.set`	Set method access to public, protected, or private.	
`UxDeclsEnableClassMode.set`	Edit the declaration of a C++ class.	*"Editing C++ Classes in the Declaration Editor"* on page 178

For more information on configuring UIM/X through resources, see the *UIM/X Installation Guide.*

Editing C++ Classes in the Declaration Editor

The Declaration Editor has two views: an Interface view and a Class view. The Interface view is the familiar Declaration Editor. It allows you to edit elements of the generated code that are common to both C and C++. For example, you use the Interface view to edit the Interface Function, which in generated C++ code is an external (non-member) function that creates an Instance of the class.The Class view gives you access to the elements of generated C++ code. You use the Class view to edit the class declaration and the definitions of the class constructor and destructor. Figure A-1 shows the Class view of the Declaration Editor.

.


Declaration Editor
View
OK
Apply
Cancel
Add to the list of base classes from which the generated class inherits.
Add your own code to the generated constructor.

```
/* Class includes */
#include "myClass.h"

/* Class specification */
class _UxCNewClass:
UxVisualInterface, public MyClass
{
   /* Generated members inserted here */
   /* User supplied members */
   protected:
           data *myData;
}
/* Constructor */
_UxCNewClass::
 _UxCNewClass(swidget UxParent, ...)
{
   this->UxParent = UxParent;
   myData = new data;
}
/* Destructor */
_UxCNewClass::
 ~_UxCNewClass()
{
   delete myData;
}
```

Figure A-1 Class View in the Declaration Editor

By default, the Class view is not available in UIM/X. To make the Class view available, uncomment the following resource specification in *uimx_directory*/`app-defaults/Uimx3_0`:

```
!Uimx3_0*UxDeclsEnableClassMode.set: true
```

This resource setting adds a View menu to the Declaration Editor. You use the View menu to switch between the Interface and Class views.

The Class View and Generated Code

The text you enter in the Class view of the Declaration Editor is ignored by the Interpreter. Everything you enter in the Class view is inserted as is into the generated code. Mistakes in the code you enter here will only be caught when you try to compile and link your application.

1. Most of the text you enter in the Class view is inserted in the generated header file. The exceptions are the definitions of the constructor and destructor, which go in the generated source file. The layout of a typical header file looks something like this:

```
// Method macros

   Method macros appear here.

// Class Includes

class _UxCLabelText: public _UxCInterface // Class Specification
{
public:
   // Constructor Function

   Your Constructor Function appears here.

   // Destructor Function

   Your Destructor Function appears here.

   // CreateSwidget Function

   Your CreateSwidget Function appears here.

   // User Defined Methods

   The methods you define appear here.
protected:

   // Widgets in the interface

   A list of objects appears here.

   // Interface-specific variables

   Interface-specific variables appear here.

   // Interface Function arguments

   Interface Function arguments appear here.

   // Callbacks and their wrappers

   Callbacks and their wrappers appear here.

   // Callback function to destroy the context

   Context destructor appears here.

private:

   // Generated Members

   Generated Members appear here.

   // User Supplied Members

   User-supplied members appear here.

};
```

Including Class Headers

Header files for C++ classes can be included in the Class Includes area. The Interpreter does not try to find or load these include files. Typically, this area is used to include the headers for classes added to the list of base classes in the class declaration.

It's worth noting that the files you include in the Class Includes area of the Class view come after the files you include in the Includes, Defines, Global Variables area of the Interfaces view.

The #include directives (and anything else) you enter in the Class Includes area go into the header file, directly above the class declaration. The header file is then included in the source file, after whatever you entered in the Includes, Defines, Global Variables area.

For example, suppose you generate the source and header files for the interface class LabelText. In LabelText.cc, you would see the following:

```
…
/******************************************************
Includes, Defines, and Global variables from the
Declaration Editor
******************************************************/
/* Content of Includes, Defines, Global Variables area */

#include "LabelText.h"
…
```

Adding Base Classes

The Class Specification area lets you edit the list of base classes in the class declaration. This allows you to derive a generated class from one or more base classes.

Any base class you add must have a constructor that can be called without any arguments. The generated code assumes that any class you list in the Class Specification area has a default constructor (that is, a constructor that can be called without arguments).

The definition of the constructor for a generated class does not specify arguments for the base classes listed in the Class Specification area:

```
   _UxCLabelText::_UxCLabelText( swidget UxParent, int anArg )
   {
   // constructor body
   }
```

Because the Interpreter does not parse anything in the Class view of the Declaration Editor, it will not recognize inheritance from the base classes listed in the Class Specification area. If any code is written within the interfaces that relies on such inheritance (eg. using a public or private variable or member function that is inherited from one of the base classes), this code must be protected from the Interpreter. To protect code from the Interpreter, enclose it in an `#ifndef UX_INTERPRETER` block.

Adding Members

The User Supplied Members area lets you add members to the generated class declaration. For example, you can use this area to add pure virtual member functions, online member functions, or overloaded constructors.

Note: The members you add in the User Supplied Members area come after the generated members. Because of this, you should add access specifiers for all members you enter here. Otherwise, your members will be private members of the class.

Adding Constructor Code

The Constructor area of the Class view gives you access to the body of a generated constructor:

```
_UxCLabelText::_UxCLabelText( swidget UxParent, int anArg )
{
      this->UxParent = UxParent;
      this->anArg = anArg;

      // User Supplied Constructor Code
      myData = new data;
}
```

Note that the code you enter in the Constructor area is inserted after the generated body of the constructor.

If you want to edit the argument list of the constructor, use the Interface view of the Declaration Editor to edit the Interface Function. The argument list of the Interface Function becomes the constructor's argument list when you generate C++ code.

Adding Destructor Code

The Destructor area of the Class view gives you access to the body of a generated destructor. For a Component, the code you enter in the Destructor area is inserted after the generated body of the destructor:

```
_UxCLabelText::~_UxCLabelText()
{
     if (UxThis && auto_destroy())
     {
         XtRemoveCallback( GetWidget(),
             XmNdestroyCallback,
             (XtCallbackProc)
                     &UxInterface::UxDestroyContextCB,
             (XtPointer) this);
     }

     if (UxThis)
     {
         DestroyInterface();
     }
     // User-Supplied Destructor Code
     delete myData;
}
```

A Subclass, on the other hand, only has a destructor if you enter some code in its Destructor area.

Note: You use the Destructor area to free any memory you allocated for your own data members. Do not try to delete the swidgets or any of the other data members UIM/X generates.

Generating Member Functions for Methods

When you generate C++ code, methods become member functions of the interface class on which they are defined. By default, a method is generated as a virtual, public member function.

The Method Editor provides four option menus for controlling how UIM/X generates the member function for a method. Figure A-2 shows the Method Editor and its option menus.

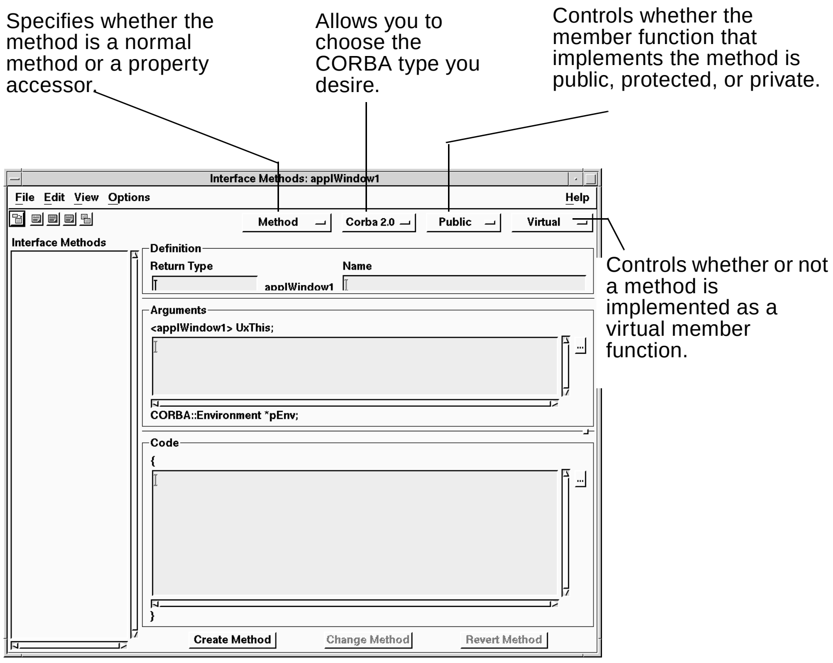

Specifies whether the method is a normal method or a property accessor.
Allows you to choose the CORBA type you desire.
Controls whether the member function that implements the method is public, protected, or private.
Controls whether or not a method is implemented as a virtual member function.
Interface Methods: applWindow1
File Edit View Options Help
Method
Corba 2.0
Public
Virtual
Interface Methods
Definition
Return Type
Name
applWindow1
Arguments
<applWindow1> UxThis;
CORBA::Environment *pEnv;
Code
{
}
Create Method
Change Method
Revert Method

Figure A-2 Method Options

By default, the Virtual and Access option menus do not appear on the Method Editor. To add these option menus to the Menu Editor, uncomment the following resource specifications in *uimx_directory*/`app-defaults/Uimx3_0`:

```
!Uimx3_0*UxMEMethodSpec.set: true
!Uimx3_0*UxMEAccessSpec.set: true
```

Choosing between Virtual and Nonvirtual

The Virtual option menu controls whether or not a method becomes a virtual member function when you generate C++ code. By default, all methods are generated as virtual members.

Typically, you set Virtual to No when you don't expect Subclasses to override the method. You may not expect the Component to be subclassed, or you may want Subclasses to inherit a mandatory implementation of the method.

If you want Subclasses to either inherit a default implementation of a method or provide their own implementation, set Virtual to Yes.

When you set Virtual to No for a method, you should not override that method in a Subclass of the Component. If you override a nonvirtual method in UIM/X, you end up redefining an inherited nonvirtual function in the generated C++ code. What this means is that the behavior of an object will depend on the declared type of the pointer that points to that object:

```
class B: public D {
public:
     void f();  // redefines nonvirtual D::f()
…
}

D d;
B *b;

b = &d;         // make b point to object of class D
b->f();         // calls B::f(), not D::f()
```

Because `b` is declared as a pointer to an object of class `B`, `b->f()` always calls the version defined by class `B`, even if `b` points to an object of class `D`.

Note: If you set Virtual to No and then override the method in a Subclass, everything will work fine in UIM/X, where the Virtual option is ignored. But there is no guarantee the generated C++ application will work properly.

Controlling Member Access

The Access option menu controls whether a method becomes a public, protected, or private member function when you generate C++ code. By default, all methods are generated as public members. Property and callback accessor methods are always public.

The Interpreter does not enforce access control for methods. The Access option controls the access attributes of member functions in generated C++ code, not of methods in UIM/X.

In UIM/X, any interface can invoke a method defined by another interface. It doesn't matter if Access is set to Protected or Private for the method. Everything will work fine in UIM/X, but the generated C++ code will not compile.

Setting the Method Type

The Method Type option menu controls whether a method is a property accessor method or a normal method. The Property Set and Property Get types are property accessors, while the Method type is either a normal method or a callback accessor.

C++ Restrictions

- Nothing in the Class view of the Declaration Editor is seen or used within the builder.
- UIM/X does not support multiple inheritance at design-time. You can, however, derive your generated interface classes from a user-defined class. See *"Adding Base Classes"* on page 182.
- You cannot provide default arguments for the member functions that implement methods. The Method Editor does not support default values for method arguments.

Frequently Asked Questions

B

Overview

This chapter provides a list of frequently asked questions (FAQ's). Most of these questions come directly from UIM/X users. This list contains only questions specifically relating to UIM/X. No attempt is made to answer questions relating to Motif.

This information is provided in the interest of enhancing user comprehension of UIM/X, helping the user avoid commonly made errors, and addressing work-around solutions to specific problems.

1. ***Do I have to pay royalties on the programs I develop?***

   No, UIM/X is sold as a development tool. Applications developed using UIM/X are exclusively the property of the developer.

2. ***Can I edit an interface file to make changes to an interface?***

   Technically, yes, you can edit interface files. However, by doing so, you increase the risk of introducing errors that UIM/X may be unable to detect. Editing interface files is strongly discouraged. If you feel compelled to edit an interface file, first be sure to save an unedited backup copy.

3. ***When I start UIM/X it complains about not being able to find include files. Why?***

   It's possible that UIM/X was not properly installed. Make sure the soft-links in `/usr/lib/X11/app-defaults` were properly created by the install procedure. `/usr/lib/X11/app-defaults/Uimx3_0` must be pointing to *`uimx_directory`*`/app-defaults/Uimx3_0`.

4. ***Why do interfaces have a different color when compiled and run than when loaded in UIM/X?***

   UIM/X sets the resource `*background` to be the same color it uses for its own interfaces. This is set in the UIM/X resource file, *`uimx_directory`*`/app-defaults/Uimx3_0`. If you want user-created interfaces to show the Motif default color while loaded in UIM/X, comment out this line.

5. ***The pixmap validator does not accept NULL or "" values. How do I disable a pixmap resource?***

   Set the value to `"unspecified_pixmap"`. This is useful when using Components and you want to specify the following as a pixmap property:

```
   DynamicPixmap ? DynamicPixmap : "unspecified_pixmap"
```

6. ***Where can I call methods from?***

   Methods can be called from other methods, callbacks, actions, and the final code section of the Declaration Editor. You can also call methods from the Property Editor to set initial property values.

7. ***If I generate code for my interfaces, for example K&R, and later generate C++ code, the linker sometimes complains about unresolved externals. Why?***

   This situation arises if code is not re-generated for all the interfaces or if not all the files are recompiled. You then have a mixture of different types of code generation.

   When switching between code generation options you should always delete any generated files and object files before re-generating the code.

8. ***When I attempt to run UIM/X under OPEN LOOK I get error messages of the type:***

```
X Toolkit Warning:
translation table syntax error:
unknown keysym name:
osfActivate
```

or:

```
X Toolkit warning:
... found while parsing '<Key>osfActivate:
ManagerGadgetSelect() '"
```

I have a valid `XKeysymDB` ***file in*** `/usr/lib/X11`**.**

The default place that OpenWindows looks for the `XKeysymDB` file is in the directory `$OPENWINHOME/lib/X11` but the `XKeysymDB` file there does not contain definitions for osf keys.

This problem only arises if the environment variable `OPENWINHOME` is set.

Make sure that LD_LIBRARY_PATH environment variable is set properly as follows:

- In C shell:

```
setenv LD_LIBRARY_PATH /usr/lib/X11:$OPENWINHOME/lib
```

- In Bourne shell:

```
LD_LIBRARY_PATH=/usr/lib/X11:$OPENWINHOME/lib
export LD_LIBRARY_PATH
```

This will ensure that the XKeysymDB file in /usr/lib/X11 will be used instead of the one in $OPENWINHOME/lib/X11

9. How can I load resources in an X-compliant way?

The X-compliant way to load resources is to use the current directory as the default and only accept explicit path names (the absolute path). This ensures that the user knows where resources are loaded from.

The X-compliant search path is as follows:

- Application class defaults:

```
/usr/lib/X11/app-defaults/
/usr/lib/X11/app-defaults/ClassName
/usr/lib/X11/$LANGUAGE/app-defaults/
(Motif only)
/usr/lib/X11/$LANGUAGE/app-defaults/ClassName
(Motif only)
```

If the XAPPLRESDIR environment variable is set then:

```
$XAPPLRESDIR/ClassName
```

otherwise:

```
~/ClassName
```

- User and system defaults:

```
~/.Xdefaults
```

If the `XENVIRONMENT` environment variable is set then:

```
$XENVIRONMENT
```

otherwise:

```
~/.Xdefaults-hostname
```

- Command-line arguments

Each resource file encountered along this path overrides any previous setting of identical resources.

Resource loading is done automatically in the main program via a call to `XtAppInitialize()`.

Note that using `UxLoadResources()` is *not* an X-compliant way of loading resources; to help maintain a semblance of compliancy, this function uses the current directory or an explicit path.

10. Why don't property values change in an object's Property Editor when I change them using the Interpreter?

Not all changes that you make in your interface result in property values being immediately updated in the object's Property Editor.

For example, suppose you set an object's `Background` color to red using its Property Editor. If you then evaluate `UxPutBackground(swidget, "blue");` in the Interpreter, the Property Editor should not change, although the object's background color should. To return the object to its initial conditions, choose Selected Objects⇒Other⇒Recreate.

11. How can I get the UIM/X Interpreter to always search a specific directory for my application's header files?

Add `-I/my/path` to the `Uimx3_0.cflags` resource.

12. ***To extend uxreaduil, the UIL compiler must be extended to recognize the new class. Does the same hold true for writing UIL?***

    The `uxreaduil` executable calls the UIL compiler internally (via `libUil.a`), thus making it necessary to extend the UIL compiler to recognize the new widget class. This is not true of `uxcgen`—since it generates UIL code but does not read it, it has no reliance on the UIL compiler. Thus, to generate UIL code for a new widget class, it is only necessary to extend `uxcgen` itself. However, the generated UIL would not be usable unless you had extended the UIL compiler to recognize the new widget class. Therefore, extending `uxcgen` is not useful unless you can also extend the UIL compiler.

Index

Index

Index

Index

Index