



UIM/X Tutorial Guide



**Integrated Computer
Solutions Incorporated**

Copyright © 2005-2007 Integrated Computer Solutions, Inc.

The *UIM/X Tutorial Guide*[™] manual is copyrighted by Integrated Computer Solutions, Inc., with all rights reserved. No part of this book may be reproduced, transcribed, stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Integrated Computer Solutions, Inc.

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

UIM/X, GUI Builder Engine, Builder Xcessory, BX, Builder Xcessory PRO, BX PRO, BX/Win Software Development Kit, BX/Win SDK, Database Xcessory, DX, DatabasePak, DBPak, EnhancementPak, EPak, ViewKit ObjectPak, VKit, and ICS Motif are trademarks of Integrated Computer Solutions, Inc.

Motif is a trademark of Open Software Foundation, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a trademark of X/Open Company Limited in the UK and other countries.

X Window System is a trademark of the Massachusetts Institute of Technology.

All other trademarks are properties of their respective owners.

Contents

Preface..... vii

Part I: Novice Mode Tutorials

Chapter 1—Building a Simple User Interface

The GUI You Will Build	3
The Steps in this Tutorial	4
Step #1: Starting UIM/X in Novice Mode	5
Step #2: Using the Novice Palette	7
Step #3: Creating the Main Window	8
Step #4: Moving and Resizing Widgets	12
Step #5: Adding the Remaining Widgets	15
Step #6: Saving Your Work	17
Step #7: Duplicating and Arranging Widgets	19
Step #8: Adding Resizing Constraints	23
Step #9: Changing Labels and Names	26
Step #10: Adding Behavior to the Push Buttons	31
Step #11: Testing the Program	34
Step #12: Generating the Code and Running the Executable	34

Chapter 2—Communicating Between Interfaces

The GUI You Will Build	38
The Steps in This Tutorial	39
Step #1: Starting UIM/X in Novice Mode	39
Step #2: Laying Out the Interfaces	41
Step #3: Saving Your Work	49
Step #4: Changing Titles, Labels, and Other Properties	51
Step #5: Adding Callbacks to the File Selection Boxes	57
Step #6: Adding Instances of the Dialogs to the Application Window	61
Step #7: Adding Items to the Menus	65
Step #8: Adding Behavior to the Menus	67
Step #9: Testing the Program	71
Step #10: Generating the Code and Running the Executable	72

Part II: Standard Mode Tutorials

Chapter 3—Creating a Drawing Editor

The GUI You Will Build	77
The Sections in This Tutorial	78

Section I: Getting Started and Drawing the Interface	78
Step #1: Starting UIM/X in Standard Mode	79
Step #2: Loading the Start-Up Project	80
Step #3: Laying Out the Working Area	81
Step #4: Changing Labels and Other Properties	87
Step #5: Adding Behavior to the Push Buttons	92
Step #6: Testing the Color-Changing Push Buttons	93
Section II: Working with Menus	94
Step #7: Adding a Pulldown Menu	95
Step #8: Adding a Cascading Menu	97
Step #9: Adding Behavior to the Color Menu	100
Step #10: Testing the Menus	104
Section III: Adding Line-Drawing Functionality	105
Step #11: Creating the Line-Drawing Push Buttons	107
Step #12: Creating the Application Window Behavior	110
Step #13: Applying the Behavior to the Line-Drawing Push Buttons	116
Step #14: Testing the Line-Drawing Push Buttons	117
Section IV: Working with Message Box Dialogs	118
Step #15: Adding the Widgets	120
Step #16: Creating Property Accessor Methods for the Message Box	122
Step #17: Adding Behavior to the Popup Push Button	124
Step #18: Testing the Message Box and Text Box	127
Section V: Generating the Application Code	127
Step #19: Customizing the Main Program and Makefile	128
Step #20: Generating the Code and Running the Executable	130

Chapter 4—Building a GUI for a Command-Line Application

The GUI You Will Build	134
The Steps in This Tutorial	135
Step #1: Starting UIM/X in Standard Mode	135
Step #2: Laying Out the Interface	136
Step #3: Changing Labels and Other Properties	141
Step #4: Adding Declarations and Final Code	143
Step #5: Adding Behavior to the Interface	145
Step #6: Testing the Program	149
Step #7: Generating the Code and Running the Executable	150

Part III: Advanced Tutorials

Chapter 5—Creating an RGB Color Editor in C++

The GUI You Will Build	155
Step #1: Starting UIM/X in Standard Mode	157
Step #2: Laying Out the Interface	157
Step #3: Changing LabelStrings and Other Properties	160
Step #4: Adding Declarations and Global Code	163
Step #5: Defining a Method to Update the Display	167
Step #6: Creating a Scale Class	168
Step #7: Exposing Properties in the Scale Class	168
Step #8: Exposing Behavior in the Scale Class	173
Step #9: Setting Properties in the Instance	175
Step #10: Adding Behavior to the Instance	177
Reordering the Connections	181
Step #11: Completing the Interface	182
Step #12: Testing the Program	185
Step #13: Generating the C++ Code and Running the Executable	187

Chapter 6—Integrating a Non-Visual Object

About This Tutorial	190
The GUI You Will Build	190
The Sections in This Tutorial	191
Section I: Creating a Non-Visual File Object	192
Step #1: Starting UIM/X in Standard Mode	192
Step #2: Creating the Non-Visual File Object	193
Step #3: Adding Functionality to the File Object	198
Section II: Using the File Object in the To Do List	203
Step #4: Loading the Start-Up Project	204
Step #5: Adding an Instance of the File Object to the Interface	206
Step #6: Modifying the To Do List Menus	208
Step #7: Testing the To Do List	211
Step #8: Generating the Code and Running the Executable	211
Section III: Integrating the File Object into UIM/X	212
Step #9: Restarting UIM/X with Builder Engine Resources	214
Step #10: Creating the New Class Code	217
Step #11: Compiling the New UIM/X Class Code	219
Step #12: Augmenting UIM/X	222
Step #13: Creating a New UIM/X Palette	223
Step #14: Polishing the Augmented UIM/X	228
Section IV: Using the Integrated File Object	231

Step #15: Starting the New Augmented UIM/X 231
Step #16: Adding a File Object to the To Do List Project 232
Step #17: Modifying the To Do List Menus 233
Step #18: Testing the Integrated Project 237
Step #19: Generating the Code and Running the Executable 237

Index 241

Preface

Overview

Welcome to the *UIM/X Tutorial Guide*, the guide to learning how to use UIM/X, the world's most powerful user interface management system. This guide introduces the basics of UIM/X in a series of step-by-step tutorials teaching the tools and techniques that will greatly assist you in developing your own applications. Whether you are new to GUI design and UIM/X, or are an experienced Motif programmer, this guide will be of service.

This guide contains all the information you need to begin using UIM/X to interactively create, modify, test, and generate code for applications with Graphical User Interfaces (GUIs). It acts as a continuing introduction to UIM/X, for those who have completed the tutorial in the *UIM/X Beginner's Guide*. It is also designed for experienced developers who want hands-on experience with advanced topics.

Tutorials are provided at three levels of experience. The first two chapters contain tutorials performed in Novice Mode, UIM/X's simplified mode for new users. The next two chapters contain more challenging tutorials in Standard Mode. The final two chapters provide instruction in advanced techniques for the experienced developer. Whatever your level of experience, you will find a tutorial in this guide to suit your needs.

Who Should Use this Guide

The *UIM/X Tutorial Guide* is intended for the user who wants to learn how to use UIM/X through hands-on experience. Though no programming skills are required to perform any of the tutorials in this guide, for a more complete understanding you should have some knowledge of C or C++, and a general understanding of the X Window System. You should also know how to use common items such as menus, Push Buttons, and Scroll Bars. If you are not familiar with these items, you might find it useful to review the *UIM/X Beginner's Guide*.

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 essential to GUI 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

Before continuing, take a moment to read the short overview that follows. After reading it you will know where to turn for the information you need.

Part I: Novice Mode Tutorials

Chapter 1, “Building a Simple User Interface”, provides an introduction to application development using UIM/X in Novice Mode. If you have never used a GUI builder such as UIM/X before, or would like an introduction to some new features, start here.

Chapter 2, “Communicating Between Interfaces”, introduces an easy-to-use technique for displaying popup dialogs such as Message Boxes and File Selection Boxes. It also explains how to use the Menu Editor to add behavior to pulldown menus.

Part II: Standard Mode Tutorials

Chapter 3, “Creating a Drawing Editor”, demonstrates how to respond to mouse action in an Application Window, and how manipulate popup dialogs using methods. As a Standard Mode tutorial, it takes advantage of the full power of UIM/X while skipping the point-and-click details of the earlier chapters. If you have previous experience with UIM/X, start here.

Chapter 4, “Building a GUI for a Command-Line Application”, demonstrates how to create an interface for a previously existing command-line application. You might want to try this tutorial, even if you will use UIM/X to build your application from start to finish, since it illustrates how to control UNIX sub-processes from within a user interface.

Part III: Advanced Tutorials

Chapter 5, “Creating an RGB Color Editor in C++”, demonstrates how to build an application using C++ and object-oriented programming techniques such as subclassing.

Chapter 6, “Integrating a Non-Visual Object”, shows how to create interface objects such as files, servers, linked lists, that by their very nature have no visual representation. It also demonstrates how to augment UIM/X, adding a new object to the executable and the Palette.

Index, a comprehensive index.

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, including Application Window and Group Box, that save you the time you might otherwise spend creating them.

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

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 <code>.login</code> file. %You have mail. Use <code>ls -a</code> to list all the files.
<i>AaBbCc12</i>	A placeholder you replace with your actual value; or words to be emphasized; or book titles.	To delete a file, type <code>rm filename</code> . You <i>must</i> be <code>root</code> to do this. See Chapter 6 in the <i>User's Guide</i> .
FileOpen	The Open option in the File menu.	Choose the FileOpen 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.

Installation Directories

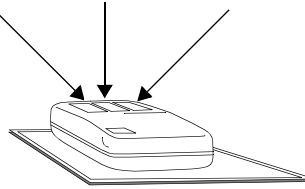
Product installation directories can depend on the platform or the user's preferences. To keep things simple, this guide uses general names for product installation directories. The following table lists the name and the corresponding product installation directory:

Name	Description
<code>uimx_directory</code>	The UIM/X installation directory

Using the Mouse

Before starting the tutorial, take a moment to review the location and usage of your mouse buttons, as illustrated in the Figure P-1 and the following table:

1: Select 2: Adjust 3: Menu




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 will 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 `app1window1` 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 User's Guide*.

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.



Part I: Novice Mode Tutorials

Overview

This section of the Tutorial Guide consists of novice-level tutorials:
Building a Simple User Interface and Communicating Between Interfaces.

Building a Simple User Interface

1

Overview

Whether you are building a complex application or a simple user interface, UIM/X dovetails with the traditional patterns of software development: you lay out the interface, add behavior, test, and generate the code. UIM/X features powerful editors that streamline this development process for maximum efficiency. Further, with Novice Mode, UIM/X makes it easier to get started building interfaces right away, even if you have no programming experience.

All the GUI building blocks you need—Push Buttons, Scrolled Windows, and so on—are stored in a Palette displayed when you start UIM/X. To lay out the interface you select the widgets you require. You simply click on the widget in the Palette, and draw (or drop) it onto the work space.

Once the widgets are in place, you are ready to change their titles, labels and other properties to customize the interface's look. In UIM/X you change properties at design time using the Property Editor. For properties such as captions that vary from widget to widget you can load widgets into the Property Editor individually. For others such as background colors, you can load multiple widgets and change properties for several widgets at once.

UIM/X features a number of other editors that facilitate interface development. The Constraint Editor, for example, allows you to add resizing constraints to your interface graphically. You can anchor elements in a fixed position, or specify their positions relative to other widgets. The result is that at runtime elements remain in proportion when the interface is resized, whether stretched by user action, or by system font changes.

In UIM/X widgets contain a great deal of built-in behavior. Menus drop down, Push Buttons push in, and so on. You can easily add advanced behavior by specifying callbacks. The callback code you write is automatically executed when the user triggers its corresponding event. For

example, a Push Button's `ActivateCallback` is activated when the user clicks on it. Other widgets have callbacks specific to their uses. Like other properties, you specify callback code using the Property Editor.

By switching to Test Mode you can verify your application's behavior without the need to generate code or leave the development environment. Once satisfied with the look and behavior of your interface, you can generate the application code for your project, in just a few clicks of the mouse.

In addition, UIM/X eases learning and lets you get started building interfaces immediately with the introduction of Novice Mode. This mode presents simplified menus and essential commands only, for a seamless growth path from first prototype to production-quality interface. Once you are familiar with the development environment you can tap into the full power of UIM/X by starting in Standard Mode.

The GUI You Will Build

This chapter demonstrates how to use UIM/X in Novice Mode to create an interface that changes colors when you click on its Push Buttons. You will create widgets using the Palette, set resizing constraints using the Constraint Editor, use the Property Editor to change properties, and add behavior using the Connection Editor. Once tested you will generate the code, then compile, link, and run the resulting application in one step.

The ColorBox interface, shown in Figure 1-1, consists of the following elements:

- *Secondary Window*: A Secondary Window widget with constraints set to allow it and its children (the other widgets in the interface) to resize gracefully.
- *Text Field*: A widget that accepts text.
- *Push Buttons*: Push Buttons that change the background color of the Text Field.

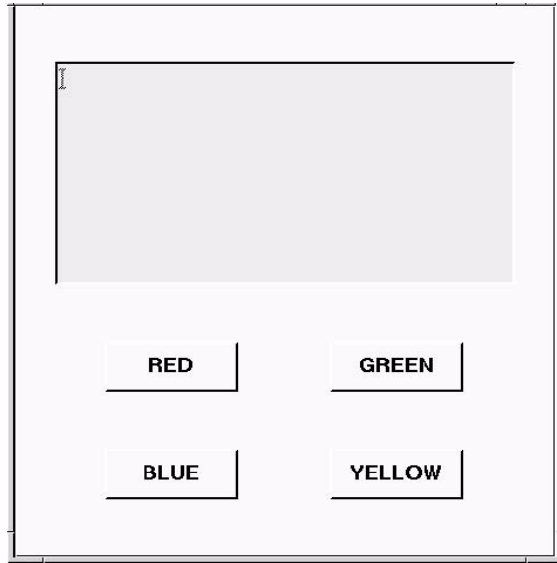


Figure 1-1 The Completed ColorBox Interface

You don't have to complete the whole tutorial in one sitting. You can stop at any point, save your work, and continue later. You do *not* need to be a programmer to understand and complete this tutorial.

The Steps in this Tutorial

This tutorial takes about 60 minutes to complete. It contains the following steps:

- Step #1: Starting UIM/X in Novice Mode
- Step #2: Using the Novice Palette
- Step #3: Creating the Main Window
- Step #4: Moving and Resizing Widgets
- Step #5: Adding the Remaining Widgets
- Step #6: Saving Your Work
- Step #7: Duplicating and Arranging Widgets

Step #8: Adding Resizing Constraints

Step #9: Changing Labels and Names

Step #10: Adding Behavior to the Push Buttons

Step #11: Testing the Program

Step #12: Generating the Code and Running the Executable

Step #1: Starting UIM/X in Novice Mode

Before you begin this tutorial, set up a new directory called `chap1`, then change to that directory, as follows:

1. Start the X Window System.
2. Open a terminal window.
3. Make a directory to store the files you will create in this tutorial:

```
mkdir chap1
```

4. Change to the directory you just created:

```
cd chap1
```

5. Start UIM/X from your new directory:

```
uimx -novice -language ansic &
```

Note: UIMX will attempt to save interfaces and project code in whatever current working directory in which the tool is started, unless the user specifies otherwise. It is easiest to begin the UIM/X session from within an existing project directory.

Note: The `-language` options instructs UIM/X to use ANSI C mode. While C++ mode accepts code written in C, for the purposes of the tutorial C mode is sufficient. By default, UIM/X starts in ANSI C mode.

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 -novice -language ansic &
```

`uimx_directory` is the base directory where you installed UIM/X.

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and UIM/X Palette appear, as shown in Figure 1-2.

6. Iconify the terminal window in which UIM/X was initially started.

Note: To restart this tutorial, begin again from step 4 above.

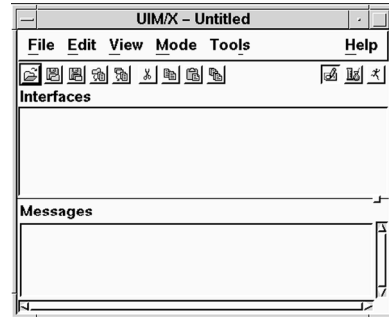
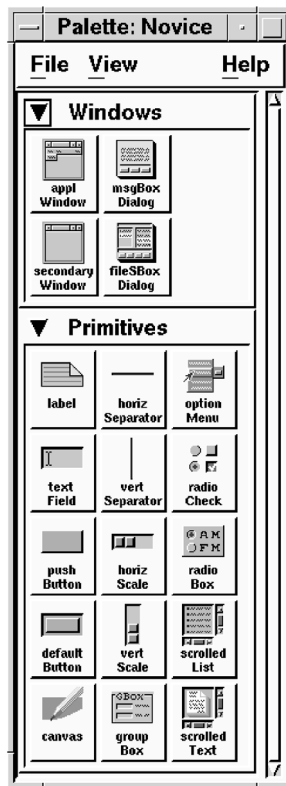


Figure 1-2 UIM/X Novice Mode Palette

Note: The project in this chapter was created using the Motif Window Manager (mwm) and its default resource values, *except* for `clientAutoPlace`, which was set to `false`. If you are using a different

window manager, or have other than default values for window manager resources, you may see slightly different object appearance and behavior from that described in this chapter.

Step #2: Using the Novice Palette

The Palette contains all the objects you use to build an interface. To create a Window or any other object in UIM/X you click on the appropriate icon in the Palette and draw it in the desired location on your screen. You can also create objects in their default size by dragging and dropping. In Novice Mode UIM/X presents a simplified Palette with fewer interface objects. In this step you will learn how to use the Novice Palette. In the next step you will use it to create a window.

1. The default view of the Novice Palette shows names and icons for all the elements on the Palette. Select View⇒By Name from the Palette menu bar.

Notice the Palette's appearance changes to show names without any icons.

2. Now select View⇒By Icon.

The Palette's appearance changes to show icons without any names. If you place the mouse cursor over an icon, bubble help appears that tells you the icon's name.

3. Finally, select View⇒By Name and Icon to show both icons and names, which is probably the most useful view for learning the package.
4. Next, notice the expand arrows to the left of the Windows and Primitives categories, as shown in Figure 1-3. Click on the expand arrow for Windows.

The category collapses to a single line, with the expand arrow pointing right to signify that it is collapsed.

5. Click on the expand arrow for Primitives to collapse both categories.

The Palette should now appear as shown in Figure 1-3.

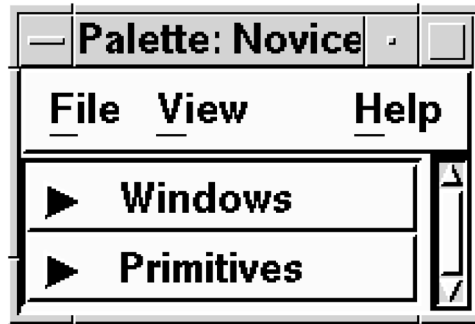


Figure 1-3 Novice Palette with Collapsed Categories

6. The Palette still takes up its original space. To save space on your desktop, choose View⇒Adjust Height from the Palette.
The Palette shrinks.
7. Select File⇒Close from the Palette. Notice how the Palette disappears from your screen.
8. Now select Tools⇒System Palette from the Project Window. The Novice Palette reappears.
9. Click on the expand arrows for Primitives, then Windows, to make each category full size again.
10. Select View:⇒Adjust Height from the Palette. The Palette returns to its full size.

Note: You can use the expand arrows to help fit the Palette on to your screen. With a few clicks of the mouse, you can collapse some categories and expand others to access only the elements you need for your project.

Step #3: Creating the Main Window

The Windows category of the Novice Mode Palette contains two windows that you use as containers for the other objects that make up your interface. An Application Window includes a menu bar, while a Secondary Window does not, and cannot. Otherwise these two windows are essentially the same.

A Window object can contain other objects, such as Push Buttons, Separators, and Scales. The Window object is referred to as the *parent* of the objects it contains, while these objects are its *children*.

To learn more about creating objects in UIM/X Novice Mode, see the *UIM/X Beginner's Guide*.

1. Check that the Design icon in the upper-right corner of the Project Window is selected, as shown in Figure 1-4.



Figure 1-4 Design Icon Selected

2. In the Windows category of the Palette, click on Secondary Window with the Select mouse button (the left one) as shown in Figure 1-5.

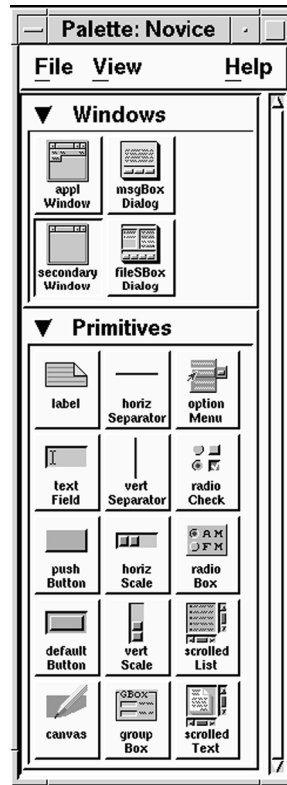


Figure 1-5 Selecting a Secondary Window from the Palette

The mouse pointer changes to a “corner” shape, representing the widget’s upper-left corner.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the Esc key. Pressing the Esc key is a convenient way to cancel drag and draw operations, for example.

3. Press and hold the Select mouse button where you want the top-left corner of the Secondary Window to be located on your screen.
4. While holding down the Select button, drag the mouse down and to the right to define the size of the new widget, as shown in Figure 1-6.

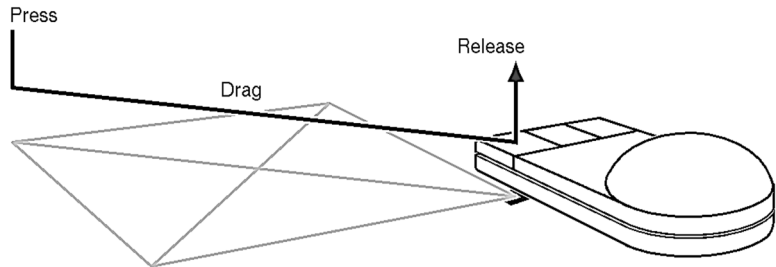


Figure 1-6 Creating a Secondary Window by Dragging and Drawing

5. Release the mouse button to complete the operation.
The Secondary Window, called `secondWindow1`, appears as shown in Figure 1-7.

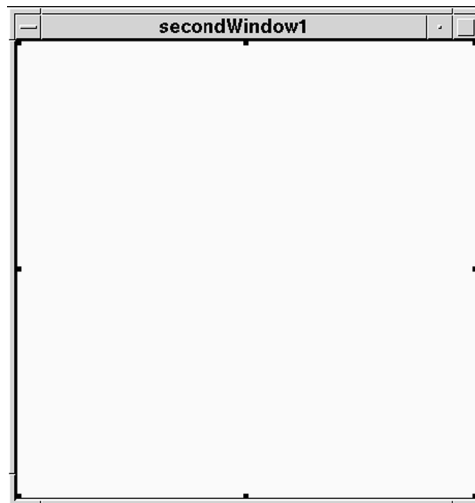


Figure 1-7 Your New Secondary Window

6. Notice that the new Secondary Window is represented by an icon in the Interfaces Area of the Project Window, as shown in Figure 1-8. Each standalone interface in a project is displayed this way, making it easy to select an entire interface, and to keep track of the interfaces in your project.

Don't worry if your Secondary Window is not the size or shape you want. You will learn how to reposition and resize it in a moment.

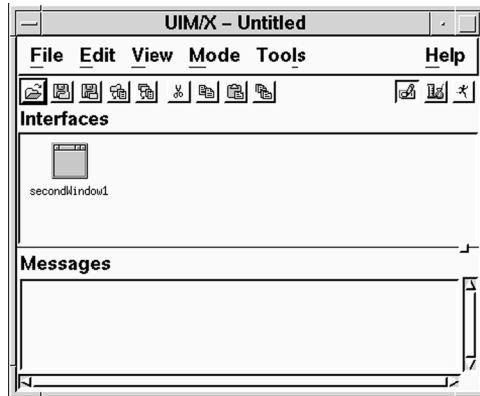


Figure 1-8 Secondary Window Icon

7. Also notice the selection handles appearing in the Secondary Window, as shown in Figure 1-9.

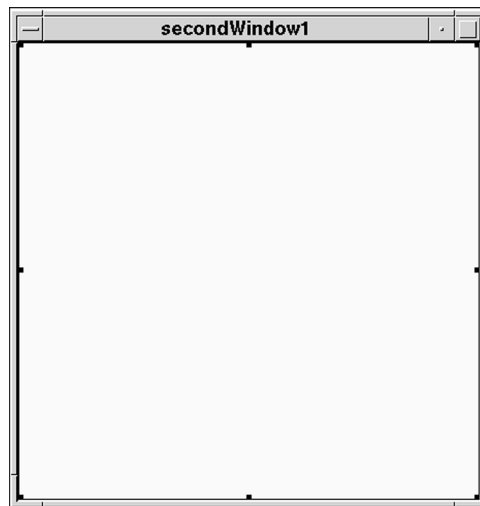


Figure 1-9 Handles Around a Selected Icon

A widget must be selected before you can move it, resize it, or change its properties. Newly drawn widgets are automatically selected.

Step #4: Moving and Resizing Widgets

It's easy to move or resize the Secondary Window widget or any other widget. When you move or resize a widget, the position of the mouse pointer is important, because UIM/X divides each widget into nine invisible regions, as shown in Figure 1-10.

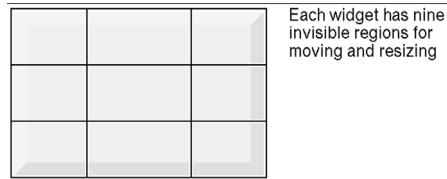


Figure 1-10 Nine Regions of a Widget



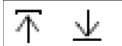
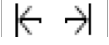
To see the nine regions of a widget, point to any corner of the widget and press the Adjust button (the middle one). The grid that appears is called the “resize grid”. Depending on which region of the widget you point to when you press the Adjust button, you will see a different resize pointer. Each resize pointer enables you to perform a different function, as listed in Table 1-1.

You can use the central region (5) of the resize grid, which displays the compass pointer, to move a selected widget to a new location. You can use the other eight regions to stretch or shrink a selected widget to a new size.



Note: Do not move or resize an interface using its window decorations (the box that appears around it). This communicates information to the window manager only, and will result in the widget returning to its original size and location at runtime.

Table 1-1 Functions of the Resize Pointers

Pointer Shape	Purpose
	Moves the widget.
	Changes the widget's height and width.
	Changes the widget's height only.
	Changes the widget's width only.

In this step you will gain some practice moving and resizing the Secondary Window widget. First you will practice moving it. Next, you will resize it. You can move and resize any selected widget the same way.

Moving the Secondary Window

In this step you will move the Secondary Window.

1. Point to the center of `secondWindow1`.
2. Press and hold down the Adjust mouse button.

Notice the mouse pointer changes to a compass shape. This means you can now move the widget.

(If you see the resize grid, release the button and try again, closer to the center of the widget.)

3. Drag `secondWindow1` to a new location and then release the button. The widget moves, as shown in Figure 1-11.

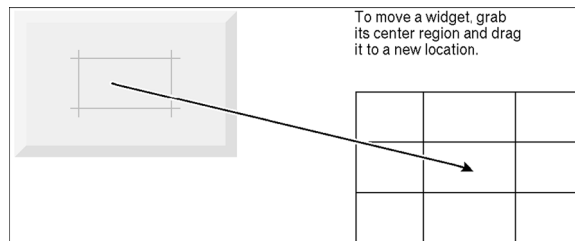


Figure 1-11 Moving a Widget

4. Repeat the process until you are comfortable moving objects around the screen.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the Esc key. Pressing the Esc key is a convenient way to cancel move operations, for example.

Resizing the Secondary Window

In this step you will resize the Secondary Window.

1. Point to one of `secondWindow1`'s resize regions, such as the lower-right corner.
2. Press and hold down the Adjust mouse button.

Notice the mouse pointer changes to a resize pointer, and the resize grid appears. This means you can now resize the widget.

(If you see the compass pointer, release the button and press again, further away from the center of the object.)

3. Drag the mouse to resize the outline of `secondWindow1` to a larger size, then release the button.

The widget reappears larger, as shown in Figure 1-12.

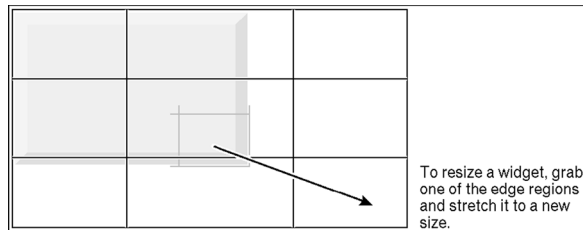


Figure 1-12 Resizing a Widget

4. Repeat the process, this time making `secondWindow1` the size you want for your interface.

Step #5: Adding the Remaining Widgets

Now that you have created the main window, you can add the remaining widgets to the interface. Dragging and dropping is a convenient way to create a widget in its default size. In this step you will add a Text Field to the interface by dragging and dropping. Next you will add a Push Button by dragging and drawing.

1. From the Primitives category of the Palette, press and hold the Adjust mouse button (the middle one) on Text Field.

The mouse pointer turns into the compass shape, and an outline of the widget appears beneath it.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the Esc key. Pressing the Esc key is a convenient way to cancel drag and drop operations, for example.

2. Place the Text Field widget in the upper half of the Secondary Window, then release the mouse button.

The Text Field appears in its default size, as shown in Figure 1-13.

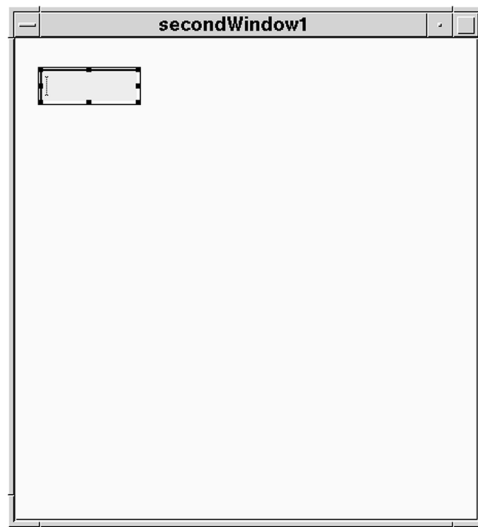


Figure 1-13 Secondary Window with Default Sized Text Field Added

3. Size and position your Text Field widget as shown in Figure 1-14

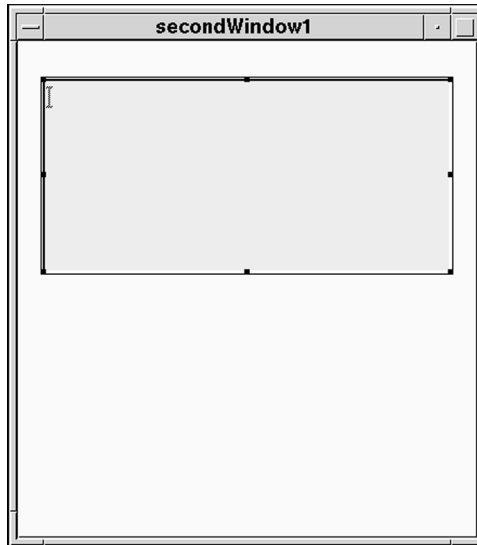


Figure 1-14 Secondary Window with Text Field Resized.

4. In the Primitives category of the Palette, click on the Push Button icon with the Select mouse button (the left one).

Notice the mouse pointer changes to a corner shape. This indicates you are ready to drag and draw the widget.

5. Move it to position the Push Button below the Text Field widget on the left, as shown in Figure 1-15.

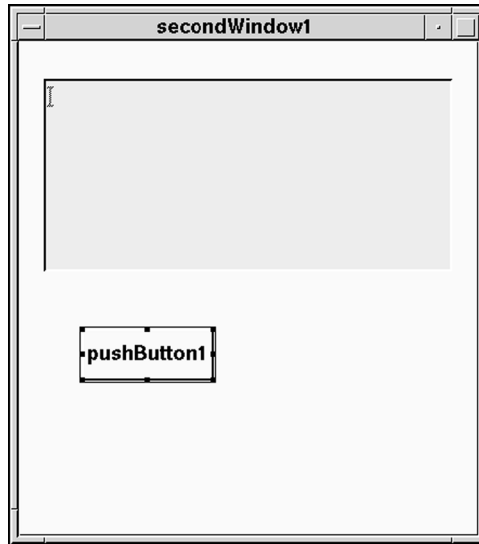


Figure 1-15 First Push Button on the Secondary Window

Note: These two widgets (automatically named `textField1` and `pushButton1`) do not appear as interface icons in the Project Window because they are children of `secondWindow1`.

Primitive widgets (such as Push Buttons) that display labels use the widget name as the default label until you change the label string using the Property Editor. You will change the label later.

Step #6: Saving Your Work

As in any software development environment, in UIM/X it is a good idea to save your work often. UIM/X facilitates the task of saving (and reloading) your interface with the notion of a project.

A project is a set of text files containing general project information and descriptions of each interface in the project. Project information is saved in a file with a `.prj` extension. UIM/X creates one project file per project. Interface information is saved in a file with a `.i` extension. UIM/X creates one interface file for each stand-alone interface in the project. The format for both of these types of files is similar to that of an X resource file.

Because this is the only format UIM/X reads, it is important to save your interface as a project even if you build your application and generate its code in one session. UIM/X loads projects by reading the project and interface files, not by reading the generated code. You need the project and interface files to make any changes to your project.

1. Select File⇒Save Project As... from the Project Window, or click on the Save Project icon in the icon bar.
2. Check that the project name selection box shows the complete path to your work directory, `chap1`, and the file name `Untitled.prj`. Click in the project file name box and replace `Untitled.prj` with `ColorBox.prj`, as shown in Figure 1-16.
3. Click on OK to save your project.
4. You can save your work at any time, in one step, by selecting File⇒Save Project or by clicking on the Save Project icon.

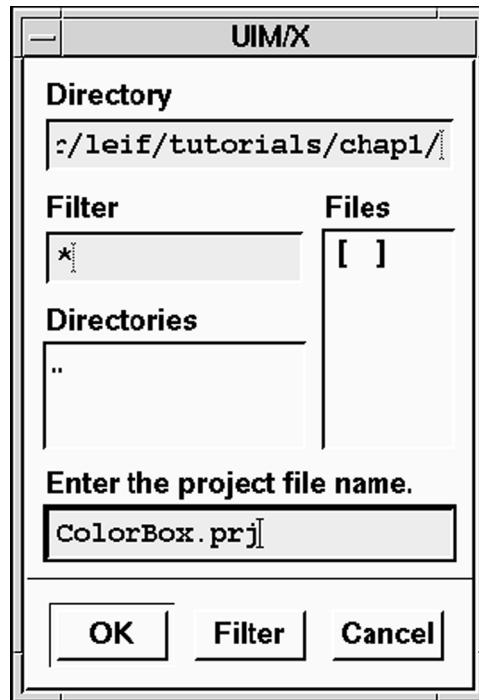


Figure 1-16 File Selection Box

Note: If you started UIM/X from the `chap1` directory as recommended, the project is saved in that directory. In the file selection box, you can also provide a complete or relative path to store the project in another directory.

Step #7: Duplicating and Arranging Widgets

In this step you will add the remaining three Push Buttons to the interface. First, you will create the Push Buttons by duplication. This ensures that all the Push Buttons are exactly the same size. Next you will arrange the Push Buttons using UIM/X's Arrange feature.

Duplicating pushButton1

In this step you will add three more Push Buttons to the interface by duplication.

1. Click on pushButton1 to select it.
2. Display the Selected Objects popup menu by pressing and holding the Menu mouse button (the right-most one) while over the Push Button.
3. The Selected Objects popup menu appears, as shown in Figure 1-17.

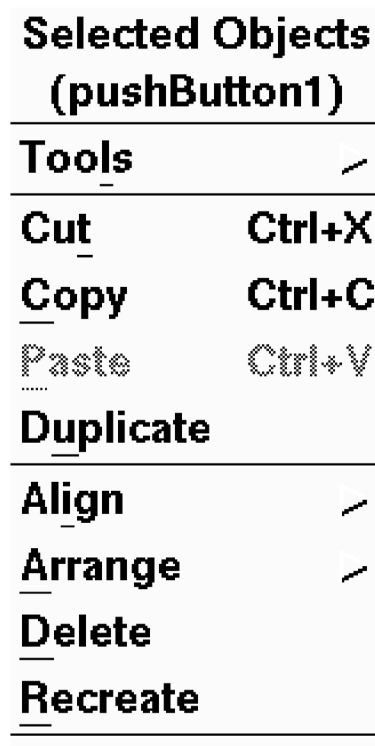


Figure 1-17 Selected Objects Popup Menu

4. Choose Selected Objects⇒Duplicate.
 A copy of pushButton1, named pushButton2, appears slightly below and to the right of pushButton1.

Note: You can also duplicate a selected object by choosing the Edit⇒Duplicate command from the Project Window, or by clicking on the Duplicate icon in the Project Window's icon bar.

5. Drag and drop `pushButton2` to its permanent location, as shown in Figure 1-18.
6. Repeat the process to create `pushButton3` and `pushButton4`. Place each of them as shown in Figure 1-18.

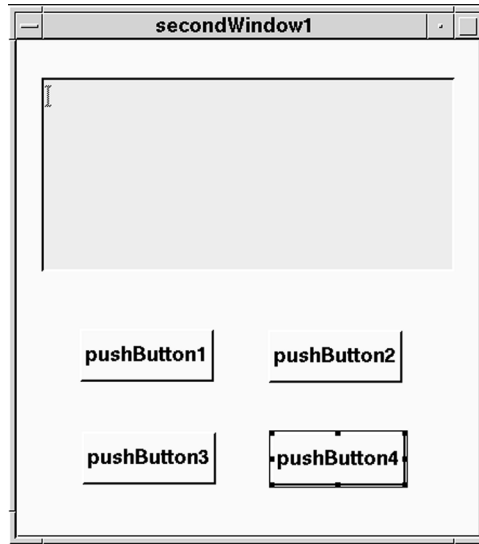


Figure 1-18 Secondary Window with all Four Push Buttons Added

7. Save your work by selecting File⇒Save Project or by clicking on the Save Project icon .

Arranging the Widgets

In this step you will use the Arrange feature to distribute the widgets within the interface. First you will select the widgets by Control-clicking. Next you will select them by marquee selection.

1. Select the `textField1`, `pushButton1`, and `pushButton3` widgets by holding down the Control key and clicking on each of the widgets in turn.
2. Deselect `pushButton4` by control-clicking it as well.
3. Press the Menu mouse button and choose Selected Objects⇒Arrange⇒(vertical arranging).

The three widgets are redrawn with equal spacing above and below each.

4. Select the `textField1`, `pushButton2`, and `pushButton4` widgets, and choose Selected Objects⇒Arrange⇒vertical arranging once more.

The widgets should now have equal vertical spacing.

5. To select the widgets by marquee selection, begin by pointing above and to the left of the Push Buttons, (but not *on* a Push Button).
6. Press and hold down the Select mouse button.

As you drag the pointer, it changes to an “O” shape, and a marquee—a dashed box—follows the pointer, as shown in Figure 1-19.

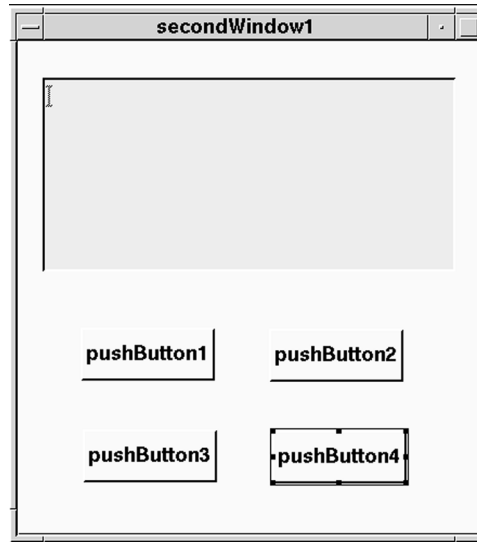


Figure 1-19 Marquee Selection

7. Continue dragging the pointer to surround `pushButton1` and `pushButton2` with the marquee.
8. Release the mouse button. Notice that the Push Buttons inside the marquee are now selected.
9. To arrange the selected widgets, press the Menu mouse button and choose Selected Objects⇒Arrange⇒horizontal arranging.
10. Now select `pushButton3` and `pushButton4`.
11. Choose Selected Objects⇒Arrange⇒horizontal arranging once more.
12. Save your work.

Step #8: Adding Resizing Constraints

You can define and apply constraints to the objects in your interface using the Constraint Editor. The Constraint Editor allows you to define constraints without having to know the numerous Motif form constraint properties. Using the Constraint Editor, you can create interfaces that maintain proportion perfectly when resized, either manually or due to a font change.

1. Click on an object within your interface and choose Selected Objects⇒Tools⇒Constraint Editor.

The Constraint Editor appears with a graphical representation of your interface.

2. Click on the Dimension icon in the Constraint Editor's icon bar.
3. Click on the bottom edge of the Text Field within the Constraint Editor.
This applies a Dimension constraint that makes the bottom of the Text Field stay a proportionate distance below the upper edge of the interface.
4. Repeat step 3 for the top edge of all four Push Buttons.
5. Now click on the Bolt icon in the Constraint Editor's icon bar.
6. Drag and draw a line from the top edge of the Text Field to the top edge of the interface.

This applies a Bolt constraint that makes the Text Field stay a set length away from the top edge of the interface. At this point all of your vertical constraints are set. The Constraint Editor should now look as shown in Figure 1-20.

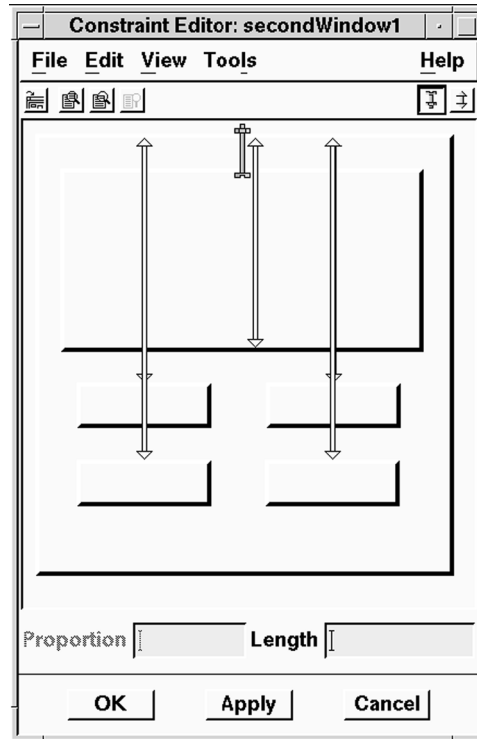


Figure 1-20 Constraint Editor Showing Vertical Constraints

7. Repeat step 6 twice more, first bolting the right edge of the Text Field to the right edge of the interface, then bolting the left edge of the Text Field to the left edge of the interface.
8. Now click on the Dimension icon in the Constraint Editor's icon bar.
9. In the same way that you did in step 3, attach a Dimension constraint to the left edge of all four Push Buttons. Then attach a Dimension constraint to the right edge of all four Push Buttons.

All of your constraints are now set. The Constraint Editor should look as shown in Figure 1-21.

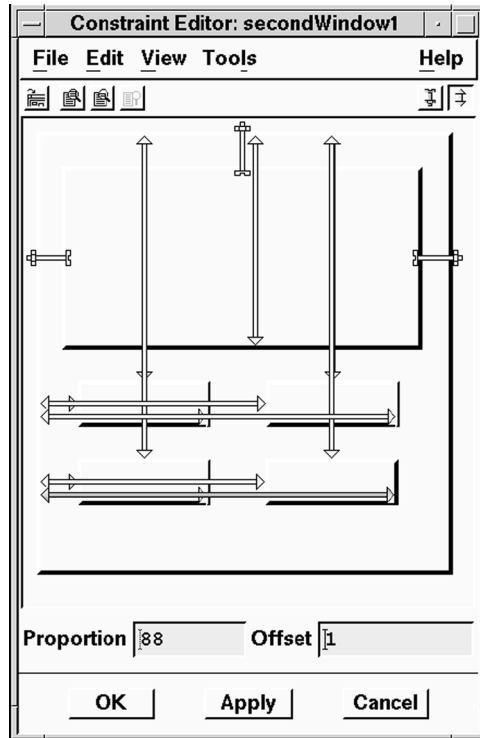


Figure 1-21 Constraint Editor Showing All Constraints

10. Close the Constraint Editor by clicking on OK.
11. Save your work.

Step #9: Changing Labels and Names

Now that the widgets are in place on the desktop, you are ready to change their labels and titles. In this step you will begin by changing each Push Button's label to match the color it will assign to the Text Field. You will also change the Secondary Window's title to something more user-friendly. In UIM/X you change properties at design time using the Property Editor.

Changing the Push Buttons' Labels

In this step you will open the Property Editor and load the first Push Button into it in one step, by double-clicking on the Push Button.

1. Double-click on `pushButton1` to open the Property Editor and load the Push Button into it.

Notice that all properties are listed in alphabetical order.

2. Scroll down the list of properties to locate the `LabelString` property.
3. Click in the text field beside the `LabelString` property and delete the default label, "`pushButton1`".
4. Give the Push Button a new `LabelString` by typing the following:
`"RED"`
 Be sure to include quotation marks around the string.
5. Your entry should look as shown in Figure 1-22.

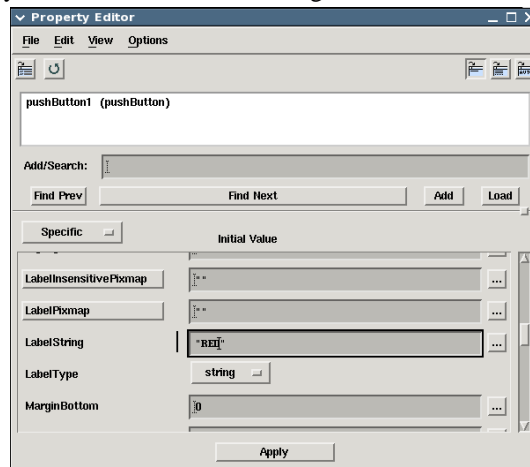


Figure 1-22 Changing the `LabelString` Property

6. Apply the change to the Push Button by clicking the Apply button at the bottom of the Property Editor.

- Note the change in appearance of the Push Button, as shown in Figure 1-23.

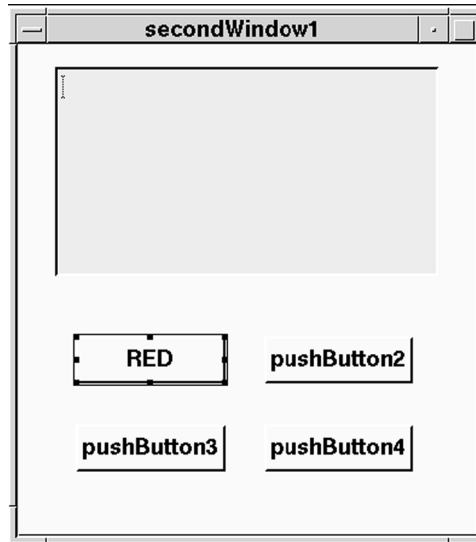


Figure 1-23 pushButton1 With New LabelString, “RED”

- Save your work.

Changing the Other Labels

You need to repeat the same process for the other Push Buttons to change their `LabelString` properties. By dragging and dropping, you can load widgets into the already open Property Editor. In this step you will drag and drop the remaining Push Buttons into the Property Editor and change their `labelString` properties.

- Move the mouse pointer to the center of the second Push Button, `pushButton2`.
- Press and hold the Adjust mouse button, just as if you wanted to move the widget.

The mouse pointer changes to the compass shape, and an outline of the Push Button appears. If the resize grid appears, press Esc and try again, closer to the center of the Push Button.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the Esc key. Pressing the Esc key is a convenient way to cancel drag and drop operations, for example.

3. Drag the outline to the Widget List area of the Property Editor, as shown in Figure 1-24, then release the mouse button.

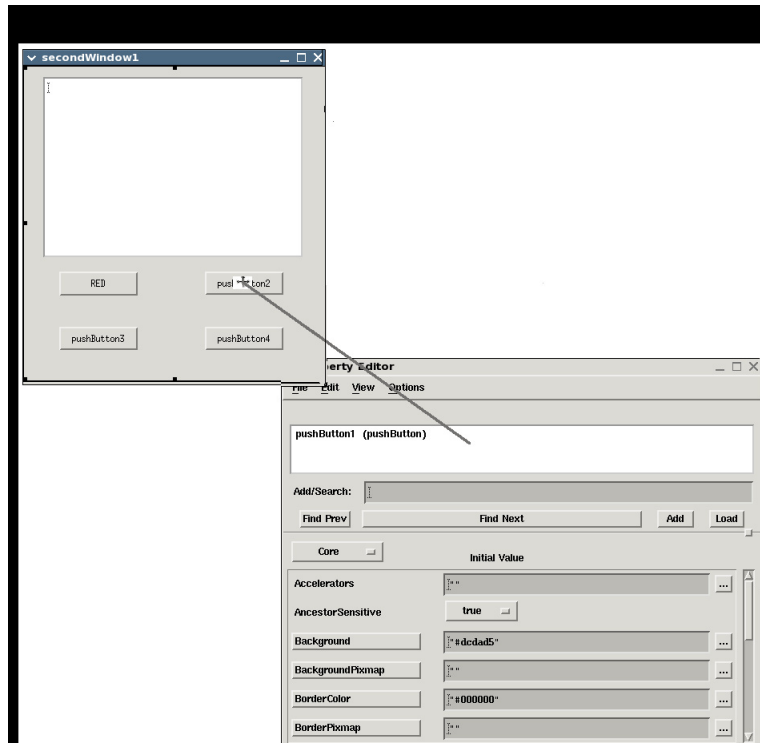


Figure 1-24 Using Drag and Drop with the Property Editor

Since the widget is too large to “fit” in the List area, be sure to move the compass shape within it.

4. Locate the `labelString` property, changing it from "pushButton2" to "GREEN".
5. Apply the changes by clicking on Apply in the Property Editor. The interface is updated to reflect the changes, as shown in Figure 1-25.

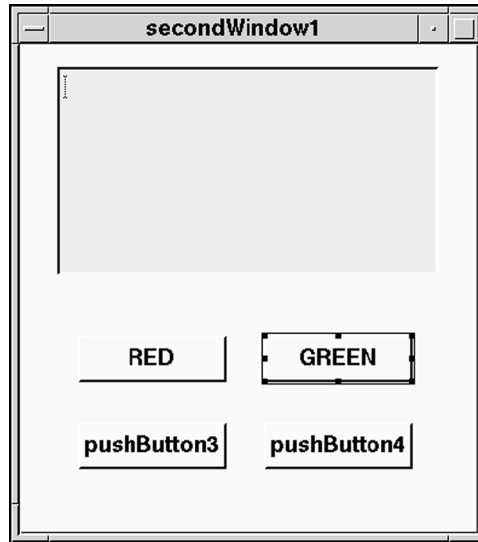


Figure 1-25 New `labelString` Property for `pushButton2`

6. Similarly, load `pushButton3` into the Property Editor by dragging and dropping (using the Adjust mouse button).
Notice the new widget replaces the last one. This is the expected Novice Mode behavior. In Standard Mode UIM/X you can set the Property Editor to replace the current widget, or add the new one to the Property Editor at the same time. You can also set the Property Editor to load widgets automatically when you select them.
7. Locate the `LabelString` property, changing it from "`pushButton3`" to "`BLUE`".
8. Apply the changes by clicking on Apply.
9. Finally, load the last Push Button, `pushButton4`, into the Property Editor.
10. Change its `LabelString` property to "`YELLOW`".
11. Apply the changes by clicking on Apply.
12. Save your work.

Changing the Secondary Window Name

One final touch is to change the name of your Secondary Window. It can be difficult to double-click on a Secondary Window, since much of it is often covered by other objects. Your Secondary Window is not covered in this way, but for practice you will load it into the Property Editor using another method: by typing its name into the Add Object area.

1. Click in the Add Object area, type `secondWindow1`, and press Return.

Your Secondary Window is loaded into the Property Editor. You can load any other widget the same way.

2. Scroll through the window's properties, and notice that both `Name` and `Title` default to `secondWindow1`.
3. Double-click in the text field for `Title`, and type in "ColorBox".
4. Click Apply. The interface is updated to reflect the changes, as shown in Figure 1-26.

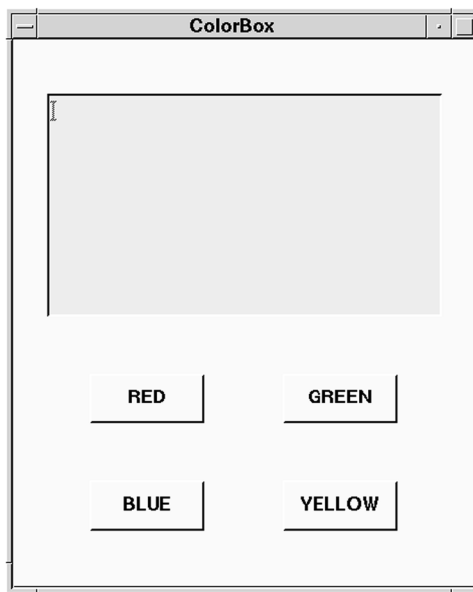


Figure 1-26 Your Interface with Labels Changed

5. Close the Property Editor.
6. Save your work.

Step #10: Adding Behavior to the Push Buttons

To simplify connecting interface elements together, UIM/X features a Connection Editor. By loading both the source and target widgets into the editor, you can view the available callbacks in the source, and the methods in the target. You can then *connect* the source's callback to the target's method visually.

In this step you will use the Connection Editor to add behavior to the four Push Buttons.

1. Select the Red Push Button, `pushButton1`, by clicking on it.
2. Press the Shift key and hold down the Select mouse button, then drag the cursor to the Text Field.

Notice a line follows the cursor, as shown in Figure 1-27. This indicates the Connection Editor is available.

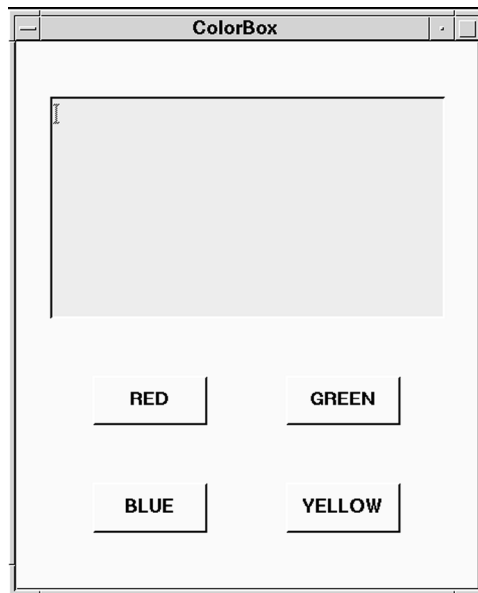


Figure 1-27 Opening the Connection Editor

3. Release the mouse button (and the Shift key) to pop up the Connection Editor, loaded with the Push Button in the Source area and the Text Field in the Target area, as shown in Figure 1-28.

Notice the Push Button's callbacks are displayed in the Callback list, and the Text Field's methods are displayed in the Method list.

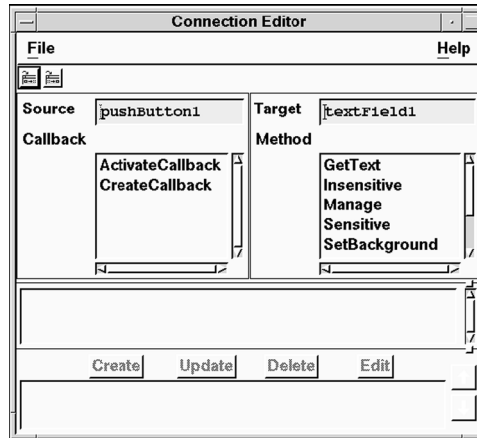


Figure 1-28 Connection Editor Loaded with `pushButton1` and `textField1`

4. Click on `ActivateCallback` in the list of callbacks, and on `SetBackground` in the list of methods.
The `Color` argument appears in the Arguments area of the Connection Editor, with a default value of `"black"`.
5. Click between the quotes and replace `"black"` with `"red"`, then click on `Create` to complete the connection.
6. The new connection appears in the Connection Editor, as shown in Figure 1-29.

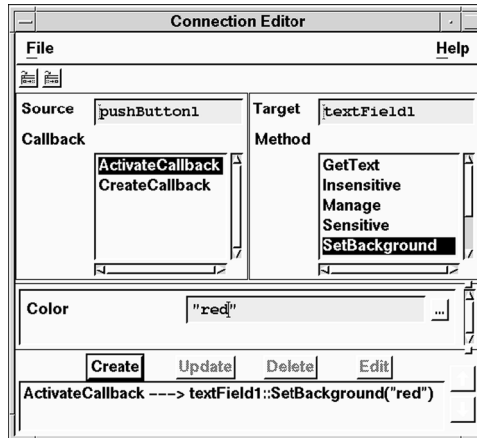


Figure 1-29 Connection Editor Showing First Connection

7. Load the Green Push Button, `pushButton2`, into the Source area in any of the following ways:
 - Click on the Green Push Button to select it, then click on the Load Source icon (the left-most one).
 - Click on the Green Push Button to select it, then choose File⇒Load Source.
 - Drag and drop the Green Push button into the Source area.
 - Type the Push Button's name, `pushButton2`, in the Source area and press Return.

Green's callbacks appear in the Callback area. Notice that `textField1` remains in the target area, its `SetBackground` method still selected.

8. Change the Color argument from "red" to "green", then click on Create to complete the connection.
9. In the same way, load the remaining Push Buttons into the Connection Editor to connect them to `textField1`.

Table 1-2 lists all four Push Buttons, with the appropriate values for Color: *Table 1-2 Values for Color* .

Table 1-2 Values for Color

Object	Color Value
pushButton1	"red"
pushButton2	"green"
pushButton3	"blue"
pushButton4	"yellow"

10. Close the Connection Editor.
11. Save your work.

Step #11: Testing the Program

Before generating code for the project in the next step, take a moment to switch to Test Mode. Test Mode allows you to see how your interface behaves, without the need to generate and compile code.

1. Click on the Test icon in the Project Window.
The Palette and any other open editors disappear. The Project Window and your interface remain.
2. Test the interface:
 - Clicking on any of the Push Buttons changes the Text widget's color.
 - Resize the interface using the window decorations. Your Push Buttons should resize and reposition elegantly.
3. When you are through, switch back to Design mode by clicking on the Design icon

Step #12: Generating the Code and Running the Executable

The final step in creating your project is to generate its code, and run the executable. UIM/X provides a convenient Run mode that allows you to run your compiled program without leaving the development environment.

In this step you will generate the code for your project, and run it, in one step.

1. Click on the Run icon in the Project Window's icon bar. The Generate Code Options window appears, as shown in Figure 1-30.

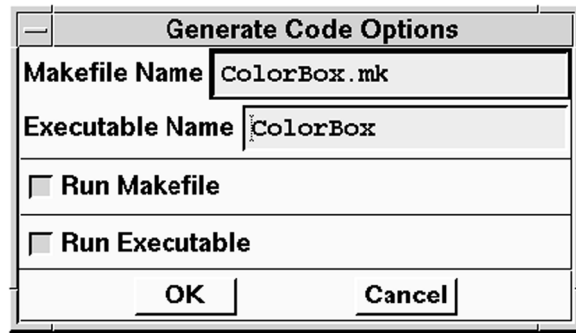


Figure 1-30 Generate Code Options Window

2. Ensure that the following radio buttons and toggle buttons are selected:
 - Run Makefile
 - Run Executable
3. Click OK.

UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
4. Test the interface. Verify that it works as it did in Test Mode.
5. Switch back to Design mode by clicking on the Design icon
6. Save your work.
7. Exit UIM/X by choosing File⇒Exit.

Communicating Between Interfaces

2

Overview

In general, an application consists of one main interface and many interfaces that pop up as a result of application and user activity. File Selection Boxes and Message Boxes are examples of commonly used dialogs that appear temporarily and behave independently of the main interface. In Novice Mode UIM/X simplifies popping up independent interfaces using *instances*. In Standard Mode, the use of instances provides even more benefits.

Instances allow you to create independent interfaces, then reuse them in other interfaces. You create the original interface exactly as you would any other, changing properties and adding behavior as desired. To reuse the interface in another, you simply drag and draw an instance of it.

Instances inherit all the properties and behavior of the original. Instances also inherit methods from the `UxVisualInterface` class, which are available for use in the Connection Editor or in callback code.

Standard Mode further extends the functionality of instances. For example, you can use the Method Editor to make properties available in the instance. By defining property accessor methods, as they are known, you can use the same instance in many interfaces, exposing properties as required. For example, you can add an instance of a File Selection Box widget to the calling interface, then use it as both the *Open* and *Save* File Selection Boxes, simply by exposing its Title property via property accessor methods.

Similarly, in Standard Mode you can create a *callback* accessor method for the original widget to make a customized callback available in the instance. By combining property and callback accessor methods, you can create interfaces with advanced built-in behavior. They can then be used as if they were local to the calling interface.

The GUI You Will Build

This chapter demonstrates how to use UIM/X in Novice Mode to create an Application Window interface that pops up dialogs. The Communication project, shown in Figure 2-1, consists of the following elements:

- *Application Window*: An Application Window widget with additional menu items added, and behavior to pop up instances of the dialogs.
- *File Selection Boxes*: Two File Selection Box widgets customized to write “open” and “save” messages to *stdout*. These pop up when the user selects File⇒Open and File⇒Save respectively.
- *Message Box Dialog*: A widget especially designed for presenting a message to the user. It pops up with a custom message when the user selects Help⇒About Application.

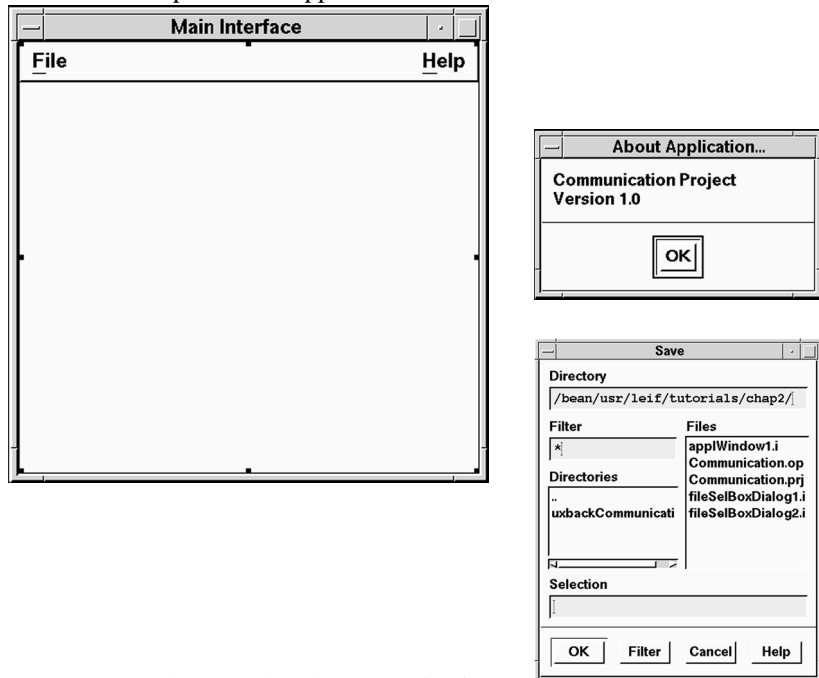


Figure 2-1 The Completed Communication Project

Note: This is a Novice Mode tutorial, designed to introduce the advantages of instances in working with dialogs. Since the Method Editor is unavailable in Novice Mode, exposing properties such as titles or message strings for use in the instance is not presented. For a Standard Mode tutorial in which properties are exposed in an instance, see Chapter 3, “Creating a Drawing Editor”. For an advanced tutorial on the same subject, see Chapter 5, “Creating an RGB Color Editor in C++”.

The Steps in This Tutorial

This tutorial takes about 60 minutes to complete. It contains the following steps:

Step #1: Starting UIM/X in Novice Mode

Step #2: Laying Out the Interfaces

Step #3: Saving Your Work

Step #4: Changing Titles, Labels, and Other Properties

Step #5: Adding Callbacks to the File Selection Boxes

Step #6: Adding Instances of the Dialogs to the Application Window

Step #7: Adding Items to the Menus

Step #8: Adding Behavior to the Menus

Step #9: Testing the Program

Step #10: Generating the Code and Running the Executable

Step #1: Starting UIM/X in Novice Mode

Before you begin building the Communication Project, set up a new directory as follows:

1. Start the X Window System.
2. Bring up a terminal window.
3. Make a directory to store the files you will create in this tutorial:

```
mkdir chap2
```

4. Change to the directory you just created:

```
cd chap2
```

5. Start UIM/X from your new directory:

```
uimx -novice -language ansic &
```

Note: The `-language` options instructs UIM/X to use ANSI C mode. While C++ mode accepts code written in C, for the purposes of the tutorial C mode is sufficient. By default UIM/X starts in C++ mode.

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 -novice -language ansic &
```

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and the Palette appear, as shown in Figure 2-2.

6. Iconify the terminal window.

Note: To restart the tutorial, begin again from Step 4 above.

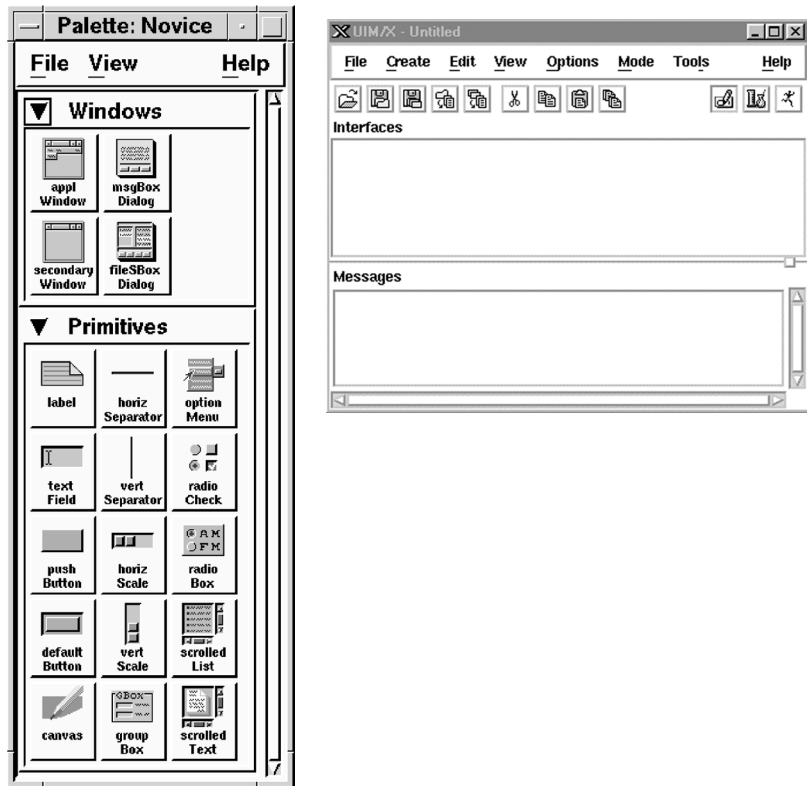


Figure 2-2 UIM/X Novice Mode Palette and Project Window

Step #2: Laying Out the Interfaces

In this step you will lay out the visual elements that make up the Communication Project interface. You will begin by drawing an Application Window, moving and resizing it if necessary. You will then add two File Selection Boxes and a Message Box to the project by drawing and duplicating.

Dragging and Drawing an Application Window

Dragging and drawing is a convenient way to create a widget of a custom size. In this step you will create the application's main interface, an Application Window, by dragging and drawing.

1. Check that the Design Mode Push Button in the upper-right corner of the Project Window is selected, as shown in Figure 2-3.



Figure 2-3 Design Mode Icon

2. In the Windows category of the Palette, click on the Application Window icon with the Select mouse button (the left one), as shown in Figure 2-4.

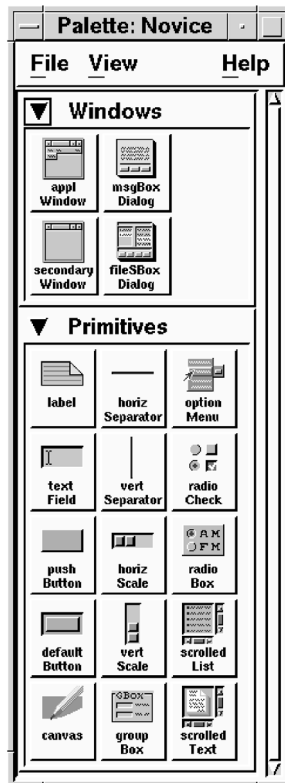


Figure 2-4 Selecting an Application Window from the Palette

Notice the mouse pointer changes to a corner shape. This indicates you are ready to drag and draw the widget.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the Esc key. Pressing the Esc key is a convenient way to cancel drag and draw operations, for example.

3. Move the mouse pointer to where you want the upper left corner of the Application Window to begin.
4. Press and hold the Select mouse button, then drag the mouse downwards and to the right to create the new widget, as shown in Figure 2-5.

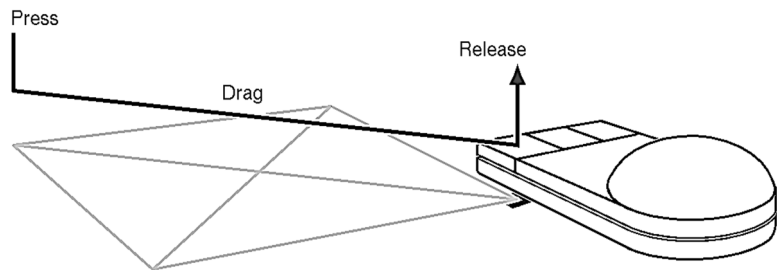


Figure 2-5 Creating an Application Window by Dragging and Drawing

5. Release the mouse button to complete the operation.

The Application Window, called `applWindow1`, appears as shown in Figure 2-6.

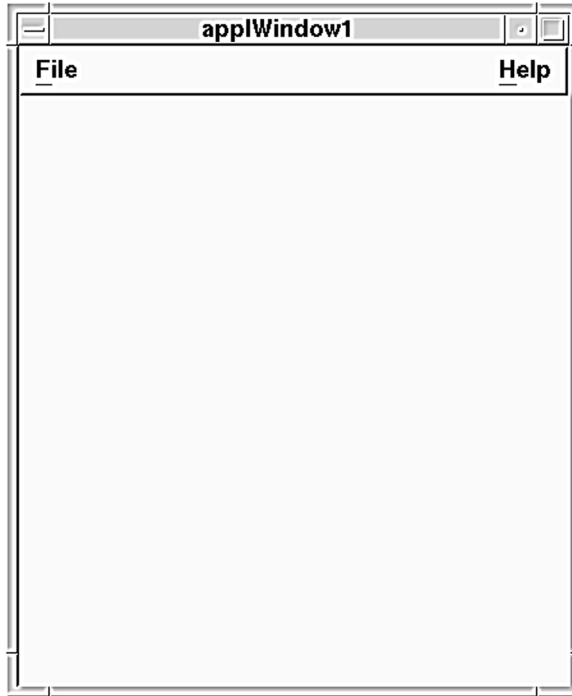


Figure 2-6 Your New Application Window Widget, `applWindow1`

6. Notice that the new Application Window is represented by an icon in the Interfaces Area of the Project Window, as shown in Figure 2-7. Each standalone interface in a project is displayed this way, making it easy to select an entire interface, and keep track of your project.

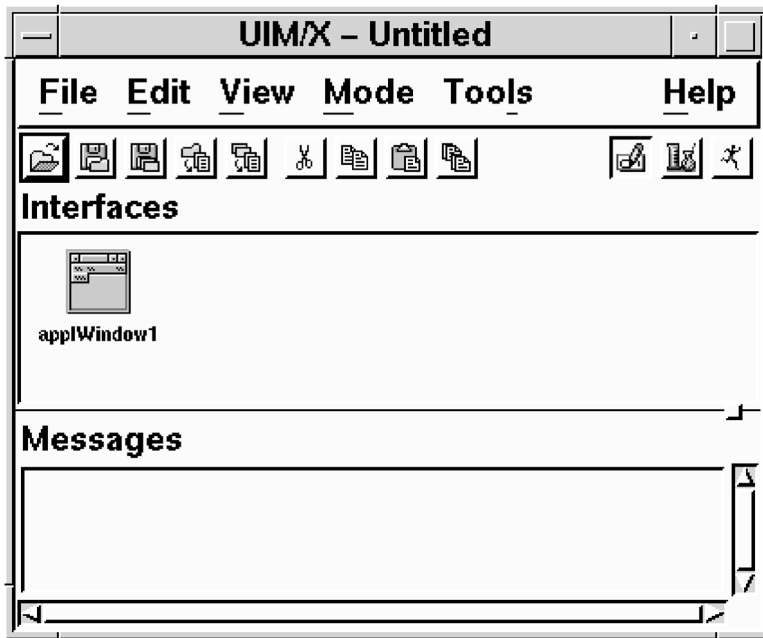


Figure 2-7 applWindow1's Icon in the Project Window

Don't worry if your Application Window is not the size or shape you want. You will learn how to reposition and resize it in a moment.

7. Also notice the selection handles appearing in the Application Window, as shown in Figure 2-8.

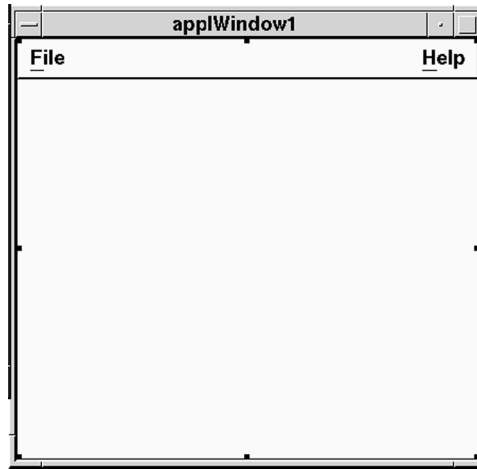
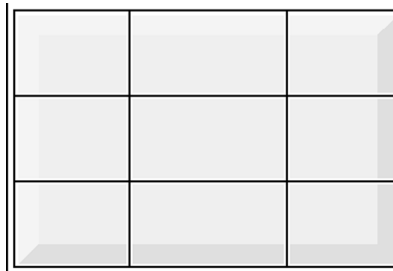


Figure 2-8 Handles Around a Selected Widget

A widget must be selected before you can move it, resize it, or change its properties. Newly drawn widgets are automatically selected.

Moving and Resizing Widgets

UIM/X simplifies moving and resizing widgets with the Resize grid. By pressing the Adjust mouse button (the center one) over a selected widget you cause the grid to appear, as shown in Figure 2-9. Depending on the position of the mouse you can stretch the component horizontally or vertically, or enlarge it in both directions at the same time. You don't even have to click on a selection handle itself, just in a region. By positioning the mouse pointer in the center square, you can move the widget without resizing it.



Each widget has nine invisible regions for moving and resizing

Figure 2-9 The Resize Grid

Each widget has nine invisible regions for moving and resizing

Note: Do not move or resize an interface using its window decorations (the box that appears around it). This communicates information to the window manager only, and will result in the widget returning to its original size and location at runtime.

Dragging and Dropping the Remaining Widgets

Dragging and dropping allows you to create widgets in their default size. In this step you will add a File Selection Box to your project, duplicate it, then add a Message Box dialog. All widgets will be dragged and dropped from the Palette.

1. Add a default-sized File Selection Box to the project by clicking and holding on the icon in the Windows area of the Palette with the Adjust mouse button (the middle one).

The mouse pointer turns into the compass shape, and an outline of the widget appears beneath it.

2. Move the outline to the desktop, then release the mouse button.

The File Selection Box appears in its default size, as shown in Figure 2-10.

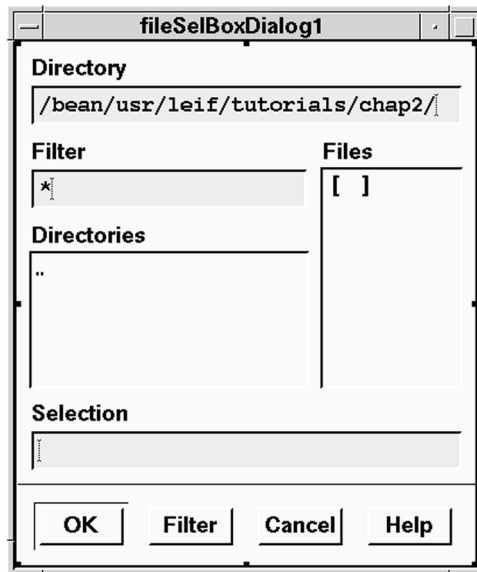


Figure 2-10 File Selection Box Added to Project

- To duplicate the File Selection Box, begin by displaying the Selected Objects popup menu by pressing and holding the Menu mouse button (the right-most one) while over the selected interface.

The Selected Objects popup menu appears, as shown in Figure 2-11.

Selected Objects (fileSelBoxDialog1)	
Tools	/
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Duplicate	
Align	/
Arrange	/
Delete	
Recreate	
Instance	

Figure 2-11 Selected Objects Popup Menu

- Select Duplicate from the Menu. UIM/X creates a new File Selection Box, fileSelBoxDialog2.
- Position the new widget beside the first by pressing and holding the Adjust mouse button while near its center.
 If a grid appears, release the mouse button, move closer to the center of the component and try again.
- Finally, add a Message Box dialog by clicking on the icon in the Windows area of the Palette with the Adjust mouse button.
- Drag the outline to the desktop then release the mouse button.

8. If necessary, use the resize grid to stretch the Message Box until it appears as shown in Figure 2-12.

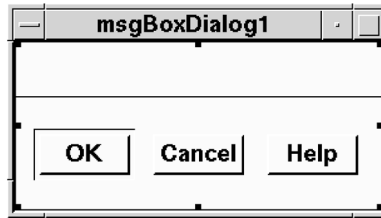


Figure 2-12 The Message Box Dialog, msgBoxDialog1

Step #3: Saving Your Work

As in any software development environment, in UIM/X it is a good idea to save your work often. UIM/X facilitates the task of saving (and reloading) your interface with the notion of a project.

A project is a set of text files containing general project information and descriptions of each interface in the project. Project information is saved in a file with a `.prj` extension. UIM/X creates one project file per project. Interface information is saved in a file with a `.i` extension. UIM/X creates one interface file for each stand-alone interface in the project. The format for both of these types of files is similar to that of an X resource file.

Because these are the only formats UIM/X reads, it is important to save your interface as a project even if you build your application and generate its code in one session. UIM/X loads projects by reading the project and interface files, not by reading the generated code. You need the project and interface files to make any changes to your project.

1. Select `File⇒Save Project As...` from the Project Window, or click on the Save Project icon in the icon bar.
2. Check that the project name selection box shows the complete path to your work directory, `chap2`, and the file name `Untitled.prj`.

Click in the selection box and replace `Untitled.prj` with `Communication.prj`, as shown in Figure 2-13.



Figure 2-13 The File Selection Box

3. Click on OK to save your project.
4. You can save your work at any time, in one step, by selecting `File⇒Save Project` or by clicking on the Save Project icon .

Note: If you started UIM/X from the `chap2` directory as recommended, the project is saved in that directory. In the file selection box, you can also provide a complete or relative path to store the project in another directory.

Step #4: Changing Titles, Labels, and Other Properties

Now that the widgets are in place on the desktop, you are ready to change their titles, labels and other properties. In this step you will begin by changing the Application Window's title to something more user-friendly. Then you will change the default string displayed in the Message Box. In UIM/X you change properties at design time using the Property Editor.

Changing the Application Window's Title

In this step you will select the Application Window and load it into the Property Editor to change its `Title` property.

1. Select the Application Window, `applWindow1`, by clicking on it with the Select mouse button.

Selection handles appear, as shown in Figure 2-14.

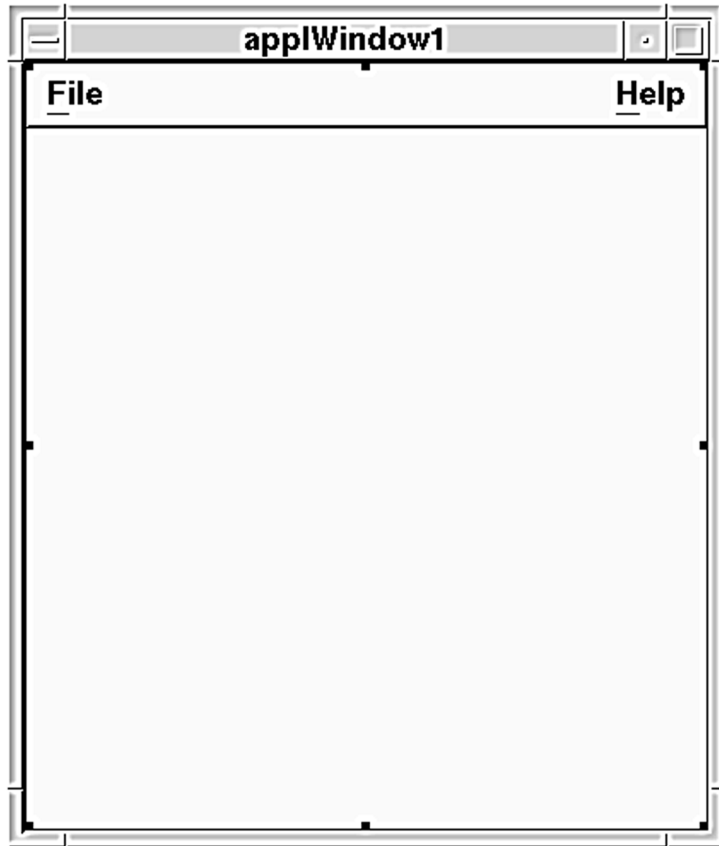


Figure 2-14 The Application Window, `applWindow1`, Selected

2. Open the Property Editor by clicking the menu mouse button (the right-most one) and choosing `Tools⇒Property Editor` from the Selected Objects popup menu.

The Property Editor appears loaded with the Application Window, as shown in Figure 2-15.

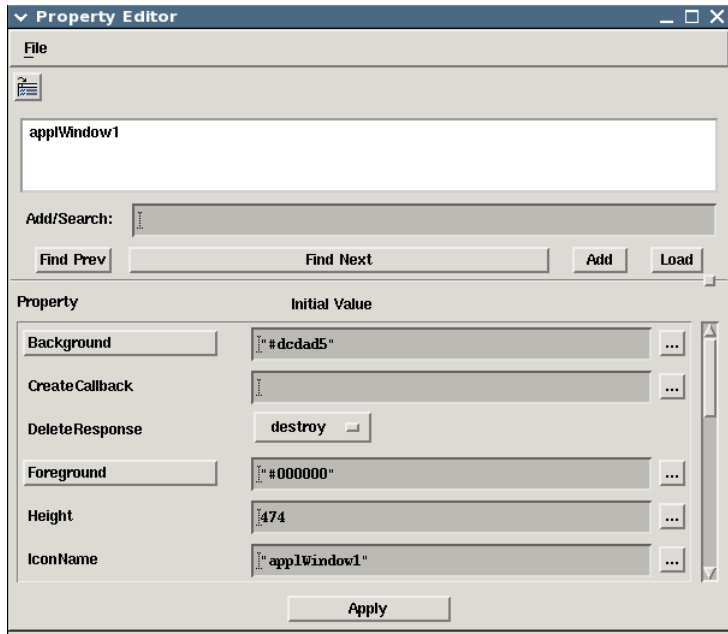


Figure 2-15 Property Editor Loaded with applWindow1

3. Locate the Title property, and replace the title "applWindow1" with the title "Main Interface".
Be sure to include quotation marks around the string.
4. Apply the change to the Application Window by clicking on the Apply button at the bottom of the Property Editor.
5. Note the change in appearance of the Application Window, as shown in Figure 2-16.

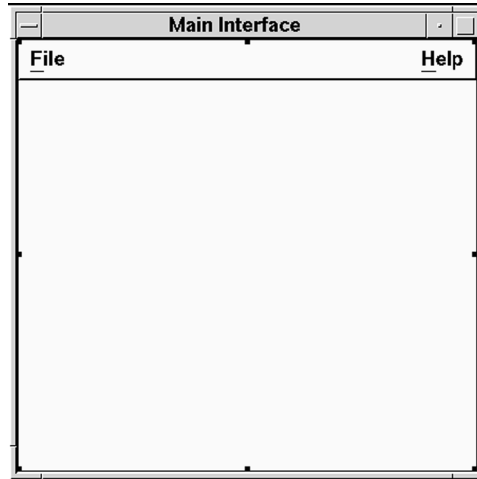


Figure 2-16 Communication Project, `applWindow1`, with New Title

6. Save your work by choosing `File⇒Save Project` in the Project Window, or by clicking on the Save Project icon .

Changing the Other Labels and Properties

You need to repeat the same process for the other widgets to change their captions and other properties. By dragging and dropping, you can load widgets into the already open Property Editor. In this step you will drag and drop the File Selection Boxes and Message Box dialog into the Property Editor and change their `labelString` properties.

1. Move the mouse pointer to the center of the first File Selection Box, `fileSelBoxDialog1`.
2. Press and hold the Adjust mouse button, just as if you wanted to move the widget.

The mouse pointer changes to the compass shape, and an outline of the File Selection Box appears. If the resize grid appears, press `Esc` and try again, closer to the center of the label.

Note: You can cancel any operation performed with the Select or Adjust mouse button by pressing the `Esc` key. Pressing the `Esc` key is a convenient way to cancel drag and drop operations, for example.

3. Drag the outline to the Widget List area of the Property Editor, as shown in Figure 2-17, then release the mouse button.

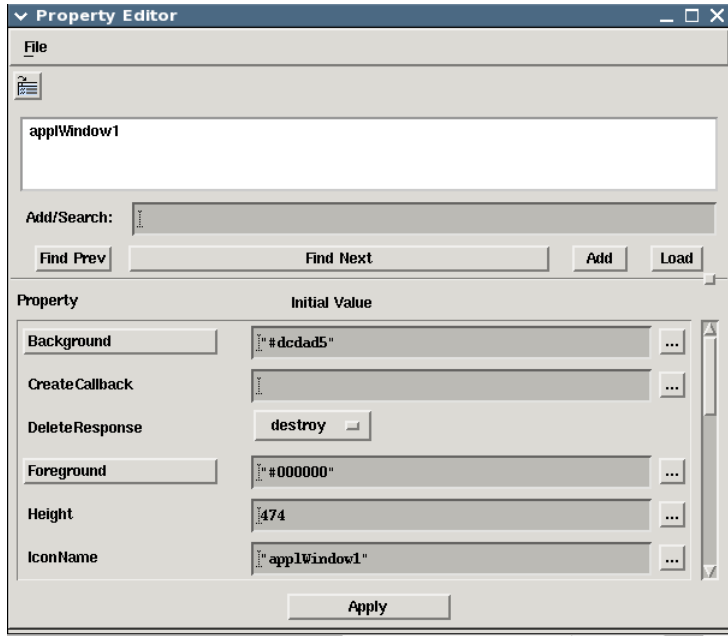


Figure 2-17 Using Drag and Drop with the Property Editor

Since the widget is too large to “fit” in the List area, be sure to move the compass shape within it.

4. Locate the `DialogTitle` property, changing it from "" to "Open".
5. Apply the changes by clicking on Apply in the Property Editor. The interface is updated to reflect the changes, as shown in Figure 2-18.

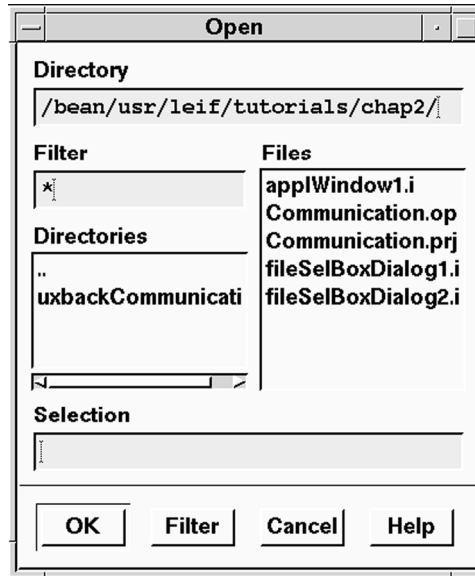


Figure 2-18 New DialogTitle Property for fileSelBoxDialog1

6. Similarly, load `fileSelBoxDialog2` into the Property Editor by dragging and dropping (using the Adjust mouse button).

Notice the new widget replaces the last one. This is the expected Novice Mode behavior. In Standard Mode UIM/X you can set the Property Editor to replace the current widget, or add the new one to the Property Editor at the same time. You can also set the Property Editor to load widgets automatically when you select them.

7. Locate the `DialogTitle` property, changing it from "" to "Save".
8. Apply the changes by clicking on Apply.
9. Finally, load the Message Box, `msgBoxDialog1`, into the Property Editor.
10. Locate the `dialogTitle` property, changing it from "" to "About Application".
11. Locate the `MessageString` property, changing it from "" to "Communication Project\nVersion 1.0".
12. Apply the changes by clicking on Apply.
13. Table 2-1 summarizes the property changes.

Table 2-1 Property Changes for Remaining Widgets

Widget	Property	Old Value	New Value
applWindow1	Title	"applWindow1"	"Main Interface"
fileSelBoxDialog1	DialogTitle	" "	"Open"
fileSelBoxDialog2	DialogTitle	" "	"Save"
msgBoxDialog1	DialogTitle	" "	"About Application"
msgBoxDialog1	MessageString	" "	"Communication Project\nVersion 1.0"

14. Save your work by choosing File⇒Save Project from the Project Window, or by clicking on the Save Project icon

Step #5: Adding Callbacks to the File Selection Boxes

In UIM/X widgets contain a great deal of built-in behavior. Menus drop down, Push Buttons push in, and so on. You can easily add advanced behavior by specifying callbacks.

The File Selection Boxes provided contain navigation and selection behavior. At runtime (and in Test Mode) you can navigate directories by clicking on directory names, set filter masks, etc. To return the name of the selected file to the calling program, you must write callbacks for the widget.

Callback code you write is automatically executed when the user triggers its corresponding event. For example, a File Selection Box's `OKCallback` is activated when the user clicks on the OK button. In a similar fashion, other widgets have callbacks specific to their uses.

One callback all widgets have in common is the `Create` callback, triggered when the widget is created. The `Create` callback is particularly useful in hiding portions of a widget provided by default, and positioning dialogs over the calling interface.

In this step you will add callback behavior to the two File Selection Boxes in the Communication project. When the user selects a file and clicks on OK the callback code will print a message to `stdout`. You will also add code

to the Message Box's `Create` callback. The code will "unmanage" the dialog's Cancel and Help Push Buttons, and position it to pop up centered over the Application Window.

Adding Callback Code to the *Open* File Selection Box

In this step you will add callback code to the *Open* File Selection Box's OK Push Button. When the user clicks on OK the callback writes a message to *stdout*, including the selected file name. In Test Mode, UIM/X captures messages to *stdout* and writes them to the Messages area of the Project Window.

1. Load the *Open* File Selection Box into the Property Editor by dragging and dropping, or by double-clicking on it with the Select mouse button.

Note: For most widgets you can open the Property Editor and load the widget into it in one move, by double-clicking on the widget.

2. Open the Callback Editor by clicking on the Text Editor button (...) beside `OkCallback`.

The Callback Editor appears, with an empty text window ready for your callback, as shown in Figure 2-19.

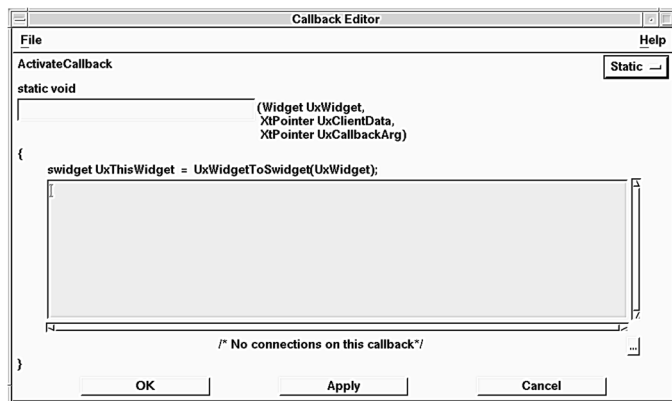


Figure 2-19 Callback Editor Loaded with `OkCallback`

3. Click in the Text Field, and type the following code exactly as it appears:

```
printf("Opening %s\n",
UxGetTextString(UxThisWidget));
```


The code prints the selected file name to *stdout*. In Test Mode it writes to the Messages Area of the Project Window. In another application you would most likely call a routine to open the selected file.

4. Click on OK in the Callback Editor.
5. Click on Apply in the Property Editor.

If you have made any typing errors, an error dialog appears, an “X” appears beside the property, and a message appears in the Messages Area of the Project Window.

Note: When you click Apply in the Property Editor your callback code is parsed and stored with the associated component, but it is not run. You use Test Mode to test your callback code.

6. Clear up any error messages that appear by checking your code against the sample provided.
 After correcting errors, be sure to click Apply in the Property Editor.
7. Save your work by choosing File⇒Save in the Project Window, or by clicking on the Save Project icon .

Adding Callback Code to the *Save File Selection Box*

In this step you will add callback code to the *Save File Selection Box*’s `OkEvent` callback. As with the *Open File Selection Box*, the callback code writes a message to *stdout*.

1. Load the *Save File Selection Box* into the Property Editor in either of two ways:
 - Select the widget then drag and drop it into the already open Property Editor.
 - Double-click on the widget.
2. Open the Callback Editor by clicking on the Text Editor button (...) beside `OkCallback`.
3. Click in the Callback Editor text field, and type the following code exactly as it appears

```
printf("Saving %s\n",
    UxGetTextString(UxThisWidget));
```

Similar to the code for the *Open File Selection Box*, the above code prints the selected file name to *stdout*. In Test Mode it writes to the Messages Area of the Project Window. In another application you would most likely call a routine to actually save the selected file.

4. Click on OK in the Callback Editor.
5. Click on Apply in the Property Editor.
6. Save your work.

Adding Callback Code to the Message Box Dialog's Create Callback

By default Message Box dialogs contain OK, Cancel, and Help Push Buttons, as shown in Figure 2-20. In this step you will hide the Cancel and Help Push Buttons by adding callback code to the Message Box dialog's Create callback. At the same time you will add code to center the dialog over the Application Window.

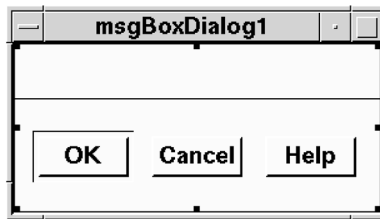


Figure 2-20 Default Message Box Dialog

1. Load the Message Box dialog into the Property Editor in either of two ways:
 - Select the widget then drag and drop it into the already open Property Editor.
 - Double-click on the widget.
2. Open the Callback Editor by clicking on the Text Editor button (...) beside CreateCallback.
3. Click in the Callback Editor text field, and type the following code exactly as it appears:


```
XtUnmanageChild(XmMessageBoxGetChild (UxWidget,
    XmDIALOG_HELP_BUTTON));

XtUnmanageChild(XmMessageBoxGetChild (UxWidget,
    XmDIALOG_CANCEL_BUTTON));

UxPutDefaultPosition( msgBoxDialog1, "true" );
```
4. Click on OK in the Callback Editor.
5. Click on Apply in the Property Editor.

Notice that the Help and Cancel Push Buttons still appear in the Message Box. This is the expected behavior. The Push Buttons will be removed when the widget is created, in Test Mode or at run time.

6. Save your work.

Step #6: Adding Instances of the Dialogs to the Application Window

When you add an instance of a dialog to an interface, it is local to the interface, and the calling interface can easily refer to it in callback code. For example, to pop up the dialog you simply call the instance's `UxManage` method.

In Standard Mode UIM/X using instances has additional advantages. By defining property accessor methods for the original, you can expose properties in the instances. The property accessor methods and the properties they make available become local to the calling interface.

In this step you will add instances of both File Selection Boxes and the Message Box to the Application Window. Next you will verify the addition using the Browser. The Browser displays an interface's widget hierarchy in a concise format, including widgets not visible at design time.

Adding the Instances

In this step you will add the instances to the Application Window.

1. Click on the *Open* File Selection Box, `fileSelBoxDialog1`, to select it.

An interface must be selected to create an instance of it.

2. Point to the Main Interface, `applWindow1`.
3. Press and hold the Menu mouse button on the Main Window interface to display the Selected Objects popup menu.

The Selected Object popup menu appears, as shown in Figure 2-21. Notice that the bottom selection is "Instance of `fileSelBoxDialog1`".

Selected Objects (pushButton1)	
Tools	/
C<u>u</u>t	Ctrl+X
C<u>o</u>py	Ctrl+C
P<u>a</u>ste	Ctrl+V
D<u>u</u>PLICATE	
A<u>l</u>ign	/
A<u>r</u>range	/
D<u>e</u>lete	
R<u>e</u>create	
I<u>n</u>stance	

. Figure 2-21 Selected Objects Popup Menu

4. Choose “Instance of fileSelBoxDialog1”. The mouse cursor changes into the corner shape.
5. Drag and draw the instance of fileSelBoxDialog1 on the Application Window, as shown in Figure 2-22.

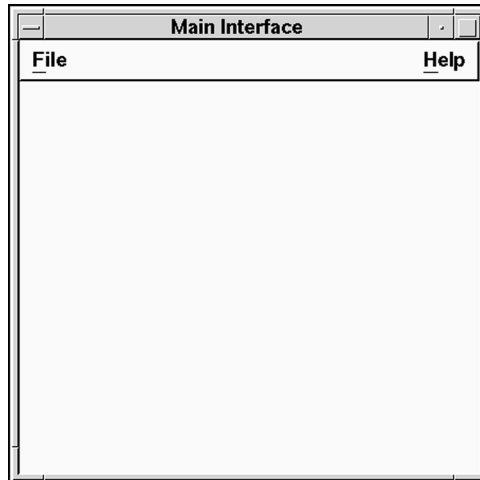


Figure 2-22 Creating an Instance of `fileSelBoxDialog1`

Note: Normally, you should draw the instance the size you want it to pop up. However, since you will be setting the sizes of the instances in the next step, it does not matter what size you draw them.

Notice that the instance is not visible on the interface. This is expected behavior for parented top-level widgets.

6. Repeat the above steps to add an instance of the *Save File Selection Box*, `fileSelBoxDialog2` to the Application Window.
7. Follow the same process to add an instance of the Message Box to the Application Window.
8. Save your work.

Verifying the Additions and Setting Display Sizes Using the Browser

UIM/X features a tool called the Browser that displays widgets in a compact format. You can view widgets by name or icon, for example, but other elements such as labels or titles, are not shown. Selecting a widget in the Browser is exactly like selecting it in the interface. Other operations, such as dragging and dropping from the Browser to an editor and

duplicating widgets are equally possible. The Browser makes it easy to work with widgets that are not visible at design time, such as instances of dialogs.

By default, instances of dialogs will pop up in the size you drew them on the interface. For precise control over an instance's display size, you should set the instance's `Height` and `Width` properties.

In this step you will open the Browser to verify that the instances were added to the Application Window. You will also use it to load the instances into the Property Editor, to set their display sizes.

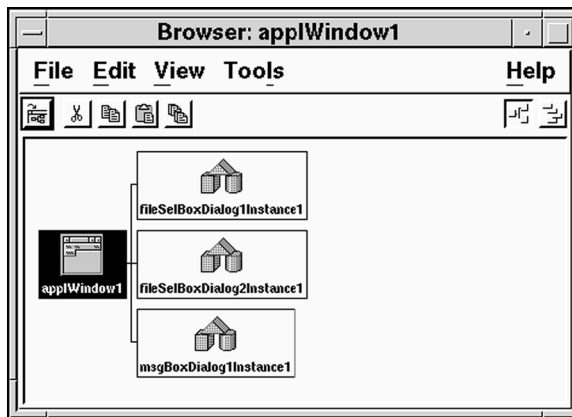
1. Select the Application Window by clicking on it with the Select mouse button.

Selection handles indicate the interface is selected.

2. Open the Browser by selecting Selected Objects⇒Tools⇒Browser.

(Recall that to display the Selected Objects popup menu, you press and hold the Menu mouse button while over the selected interface.)

The Browser appears, loaded with the Application Window, as shown in Figure 2-23.



. Figure 2-23 Browser Loaded with the Application Window

3. Choose the view you prefer using the View menu.
 - You can view the interface's widgets by name, icon, or both name and icon.
 - You can display the structure of your interface in tree or outline format.

4. Scroll the Browser window to locate the *Open* File Selection Box instance, `fileSelBoxDialog1Instance1`.
Following the UIM/X naming conventions for widgets, the instances are named `fileSelBoxDialog1Instance1`, `fileSelBoxDialog2Instance1`, and `MsgBoxDialog1Instance1`. In another application you might rename the instances to something shorter. For the purposes of the tutorial the default names will suffice.
5. Select the widget by clicking on representation in the Browser. Reverse video indicates the widget is selected.
6. Load the widget into the Property Editor by choosing Selected Objects⇒Tools⇒Property Editor.
7. Locate the *Height* property, changing it to 425.
8. Locate the *Width* property, changing it to 300.
9. Click on Apply.
10. Load the *Save* File Selection Box instance, `fileSelBoxDialog2Instance1`, into the Property Editor by dragging and dropping from the Browser.
11. Similarly, change its *Height* and *Width* properties to 425 and 300 respectively.
12. Click on Apply.
13. Finally, load the Message Box dialog instance, `MsgBoxDialog1Instance1`, into the Property Editor, and change its *Height* and *Width* properties to 150 and 285 respectively.
Be sure to apply your changes.
14. Close the Property Editor by choosing File⇒Close in the Property Editor.
15. Close the Browser by choosing File⇒Close from the Browser menu.

Step #7: Adding Items to the Menus

In UIM/X building menus is simplified for two main reasons. First, menu elements contain built-in behavior including automatic resizing and positioning. You never have to worry about the size of menu labels, or pull-down behavior, for example. Second, UIM/X features an editor called the Menu Editor that provides structured access to your menus.

By default Application Windows are provided with a menu bar and two pulldown menus: a File menu and a Help menu. In this step you will use the Menu Editor to add an Open and a Save item to the File menu.

1. Open the Menu Editor by selecting the main interface, `applWindow1`, and choosing Selected Objects⇒Tools⇒Menu Editor.

The Menu Editor appears, as shown in Figure 2-24.

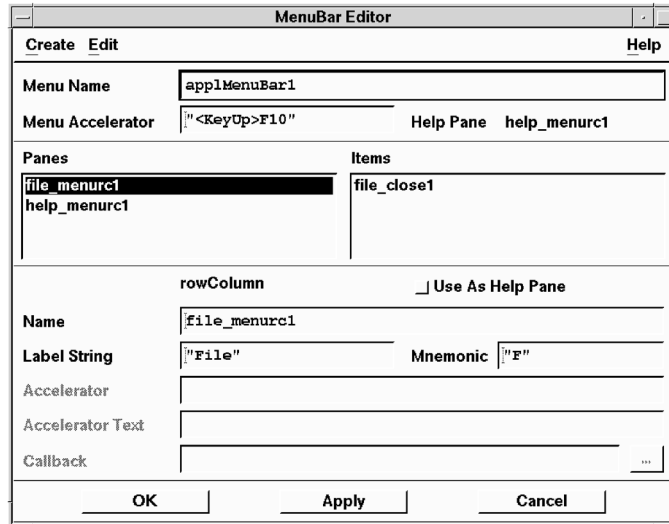


Figure 2-24 Menu Editor

2. Rename the File menu's default item by clicking on `file_close1` in the Items list, and entering the values shown in Table 2-2.

Table 2-2 Property Values for Open Menu Item

Property	Value
Name	openButton
Label String	"Open"
Mnemonic	"O"

In Novice Mode, items are always added *after* an existing item. Since this project features an Exit item at the end of the menu, you must reassign the default item provided. In Standard Mode you can add items before *or* after existing ones.

3. Apply the change by clicking on Apply.
4. To add an item to the menu ensure `openButton` is selected in the Items list, then choose Create⇒Item⇒Push Button.
5. Enter the values shown in Table 2-3 for the new Push Button:

Table 2-3 Property Values for Save Menu Item

Property	Value
Name	saveButton
Label String	"Save"
Mnemonic	"S"

6. Apply the change.
7. Add another item to the menu by selecting Create⇒Item⇒Push Button.
8. Enter the values shown in Table 2-4:

Table 2-4 Property Values for Close Menu Item

Property	Value
Name	exitButton
Label String	"Exit"
Mnemonic	"x"

9. Apply the change.
10. Save your work.

Step #8: Adding Behavior to the Menus

To simplify connecting interface elements together, UIM/X features a Connection Editor. By loading both the source and target widgets into the editor, you can view the available callbacks in the source, and the methods in the target. You can then *connect* the source's callback to the target's method visually, rather than via callback code.

In this step you will use the Connection Editor to add behavior to the File menu's Open, Save, and Exit items, and the Help menu's About item using the Connection Editor.

Opening the Connection Editor and Making the First Connection

In this step you will load the File menu's Open item into the Connection Editor, and connect it to the *Open* File Selection Box instance using the instance's *Manage* method.

1. Select the Open menu item, `openButton`, by clicking on it in the Menu Editor.

- Open the Connection Editor by selecting Edit⇒Connection From⇒Item in the Menu Editor.

The Connection Editor appears loaded with `openButton` in the Source area, as shown in Figure 2-25. Notice the `openButton`'s callbacks are listed in the Callback area of the Connection Editor.

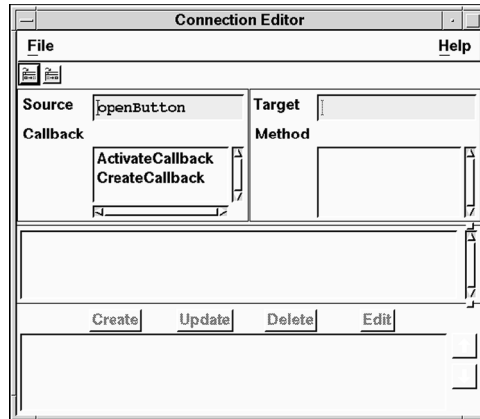


Figure 2-25 Connection Editor

- Open the Browser by choosing Selected Objects⇒Tools⇒Browser while over the Main Interface.
- In the Browser, locate the instance of the *Open* File Selection Box, `fileSelBoxDialog1Instance1`.
- Drag and drop it from the Browser to the Target area of the Connection Editor (using the Adjust mouse button). You can also load an object into the Connection Editor by selecting it, then clicking on the Load Target icon (the right-most one).

The instance's default methods are listed in the Method area of the Connection Editor, as shown in Figure 2-26.

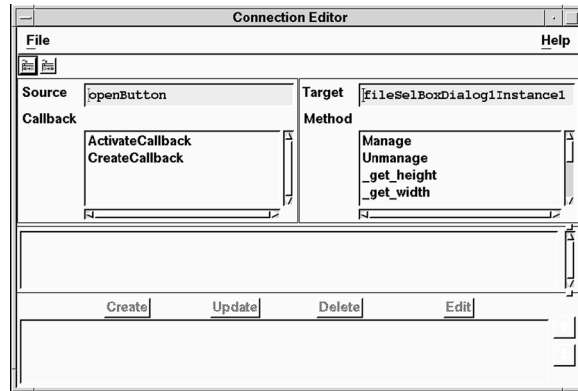


Figure 2-26 Connection Editor Showing *Open*'s Methods

6. Click on `ActivateCallback` in list of callbacks, and on `Manage` in the list of methods.
Any parameters or return values available appear in the parameters area.
7. Complete the connection by clicking on `Create`.
8. The new connection appears in the Connection Editor, as shown in Figure 2-27.

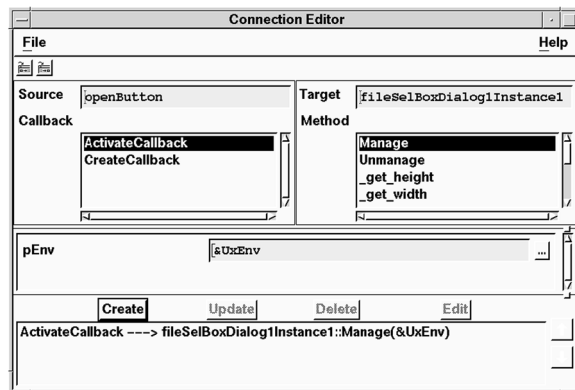


Figure 2-27 Connection Editor Showing New Connection

9. Save your work.

Making the Remaining File Menu Connections

In this step you will create the remaining File menu connections from menu items to the dialogs they pop up. First you will connect the File menu's Save item to the *Save File Selection Box*. Then you will connect the File menu's Close item to the Application Window's Exit method.

1. Click on the `saveButton` item in the Menu Editor, and choose `Edit⇒Connection From⇒Item`.

The `saveButton` is loaded into the Connection Editor.

2. Drag and drop the instance of the *Save File Selection Box*, `fileSelBoxDialog2Instance1`, from the Browser to the Target area of the Connection Editor.

Notice that `ActivateCallback` and `Manage` remain selected in the Connection Editor.

3. Complete the connection by clicking on `Create`. The new connection is added to the list.

4. Click on the `exitButton` item in the Menu Editor, and choose `Edit⇒Connection From⇒Item`.

The `exitButton` is loaded into the Connection Editor.

5. Drag and drop the Application Window itself, or its representation in the Browser, to the Target area of the Connection Editor.

Note that `ActivateCallback` remains selected in the Source Callback list.

6. Create the connection to exit the application by clicking on `Exit` in the Target Method list.

7. Complete the connection by clicking on `Create`.

8. Save your work.

Making the Help Menu Connection

In this step you will connect the Help menu's About item to the Message Box dialog.

1. Click on the Help menu pane, `help_menurc1`, in the Menu Editor.
2. Select the About item, `help_about1`, by clicking on it.
3. Load it into the Connection Editor by choosing `Edit⇒Connection From⇒Item` in the Menu Editor.
4. Drag and drop the Message Box dialog, `msgBoxDialogInstance1`, from the Browser into the Target area of the Connection Editor.

Note that `ActivateCallback` remains selected in the Source Callback list.

5. Click on `Manage` in the Method area.
6. Complete the connection by clicking on `Create`.

The new connection appears in the Connection Editor, as shown in Figure 2-28.

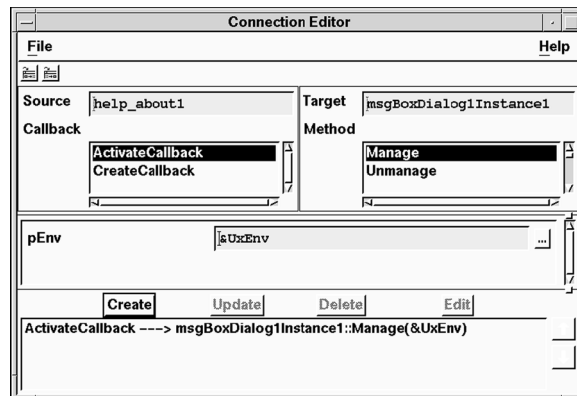


Figure 2-28 Connection Editor Showing Help Menu's *About* Connection

7. Close the Connection Editor by choosing `File⇒Close`.
8. Close the Menu Editor (by choosing `Cancel`) and the Browser (by choosing `File⇒Close`).
9. Save your work.

Step #9: Testing the Program

Before generating code for the project in the next step, take a moment to switch to Test Mode. Test Mode allows you to see how your interface behaves, without the need to generate and compile code.

1. Hide the *Open*, *Save*, and *About Application* interfaces by selecting them in the Project Window and choosing `Selected Interfaces⇒Hide`.
2. Click on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

3. Test the popup dialog behavior:
 - Choosing File⇒Open or File⇒Save pops up the *Open* and *Save* File Selection Boxes respectively.
 - Choosing Help⇒About Application pops up the Message Box dialog.
 - Clicking on OK or Cancel in the dialogs pops down the interface.
4. Test the other added behavior:
 - Clicking on OK in either File Selection Box writes a message to the Messages Area of the Project Window. At runtime, it writes the message to *stdout*.
 - Choosing File⇒Exit causes a dialog to pop up stating that the `exit()` function was called. At runtime it terminates the application.
5. When you are through, switch back to Design Mode by clicking on the Design icon

Step #10: Generating the Code and Running the Executable

The final step in creating your project is to generate its code, and run the executable. UIM/X provides a convenient Run Mode that allows you to run your compiled program without leaving the development environment.

In this step you will generate the code for your project, and run it, in one step.

1. Click on the Run icon in the Project Window's icon bar.
2. The Generate Code Options window appears, as shown Figure 2-29.

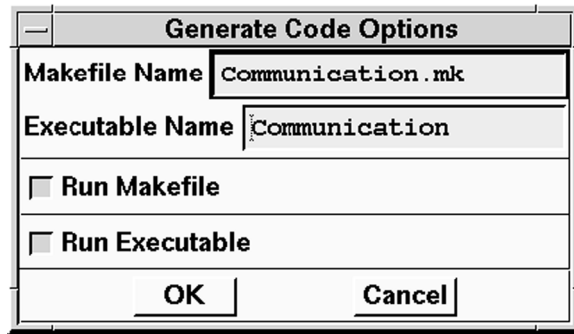


Figure 2-29 Generate Code Options Window

3. Ensure that the following radio buttons and toggle buttons are selected:
 - Run Makefile
 - Run Executable
4. Click OK.

UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
5. Test the interface. Verify that it works as it did in Test Mode.
6. To stop the program, choose File⇒Exit from the Application Window.
7. Switch back to Design mode by clicking on the Design icon
8. Save your work.
9. Exit UIM/X by choosing File⇒Exit in the Project Window.

Where to Go From Here...

As noted in the introduction, this tutorial presented some of the advantages of using instances when working with dialogs. Since it was a Novice Mode tutorial, however, it could not explore all the advantages Standard Mode has to offer.

For example, this tutorial featured two File Selection Boxes with separate titles (“Open” and “Save”) that printed custom messages to *stdout*. In Standard Mode only one File Selection Box would have been required. By defining property accessor methods via the Method Editor, you could expose the related properties in the instance. Then, when popping up the instance, you could set the title appropriately, via callbacks, or graphically

using the Connection Editor. Similarly you could have created a method associated with the File Selection Box's OK Push Button. UIM/X presents appropriately named methods in the Behavior category for the instance.

As noted, for a Standard Mode tutorial in which properties are exposed in an instance, see Chapter 3, "Creating a Drawing Editor". For an advanced tutorial on the same subject, see Chapter 5, "Creating an RGB Color Editor in C++".

Note: Consult the *Release Notes* for information about the currently supported C++ code development.

Part II: Standard Mode Tutorials

Overview

The tutorials in this section will be completed in Standard Mode of UIM/X. These tutorials are Creating a Drawing Editor and Building a GUI for a Command-Line Application.

Creating a Drawing Editor

3

Overview

In most applications there are times when you want a user action to change a property at runtime. For example, in a color editor, you might want to allow users to click on a color chip and assign it to an object. In UIM/X you can use callbacks to activate the runtime property change, and the `UxGetProperty` and `UxPutProperty` functions to get and assign property values respectively.

UIM/X simplifies the task of creating menus with an easy-to-use Menu Editor, providing all the widgets you need and a consistent builder interface whether you are designing pulldown, popup, or cascading menus. Built-in behavior includes automatic resizing of menu panes to fit their captions, positioning and pulldown behavior. Assigning keyboard accelerators and mnemonics is a snap.

The working area of your interface is often referred to as the application window, because user activities—such a mouse action or keyboard clicks—call the functions in your underlying application. UIM/X lets you quickly create application window behavior by providing structured access to your application's functionality via translation tables, where you pair mouse or keyboard events with your application's actions.

UIM/X provides a number of dialog widgets to support communication with the user. The Message Box is the easiest way to present a simple message, while the Prompt lets you obtain a *yes* or *no* answer, for example. While their purposes differ, all dialog widgets share an ease of use, and the potential for quick customizing. By writing an interface method for your chosen dialog widget, you can customize it to pop up from any interface in your project, with messages created on-the-fly.

The GUI You Will Build

In this chapter you will create an interface to function as a Drawing Editor. The Drawing Editor illustrates how to change properties at runtime, change mouse behavior in the application window, build menus, and pop up a dialog.

The completed interface, shown in Figure 3-1, consists of the following areas:

- *Menu Bar*: Contains pull-down menus with regular panes, mutually exclusive panes, and panes for selecting options.
- *Color-Changing Push Buttons*: Push Buttons that change the background color of the work area.
- *Line-Drawing Push Buttons*: Allow the user to select from a line, rectangle, circle, or ellipse, then draw the shape in the work area.
- *Work Area*: Contains a Frame in a Scrolled Window where the user can draw shapes using the mouse.
- *Popup Dialog Area*: Clicking on the Push Button pops up a Message Box displaying any text entered in the Text field.

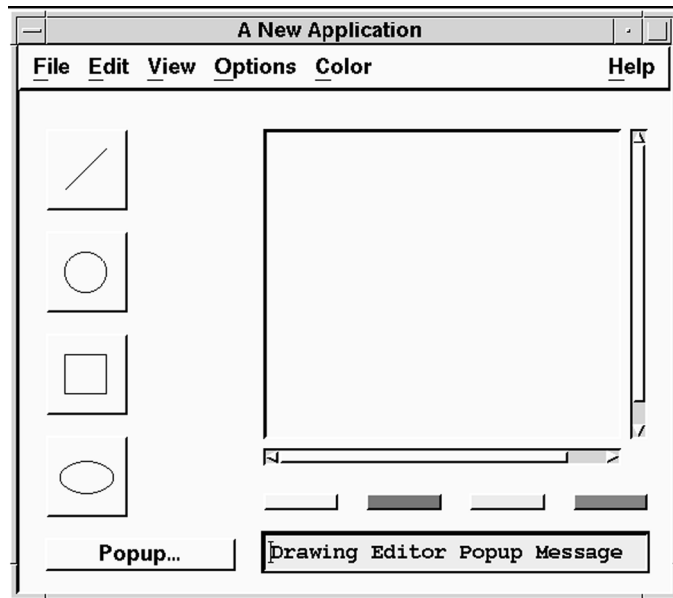


Figure 3-1 The Completed Drawing Editor Project

To allow the tutorial to focus on new skills while presenting as many features as possible, a start-up project has been provided. It includes the main window, with several working menus. Exploration of the callback code provided is left to the reader.

The Sections in This Tutorial

This tutorial takes about 120 minutes to complete. It contains the following sections:

Section I: Getting Started and Drawing the Interface

Section II: Working with Menus

Section III: Adding Line-Drawing Functionality

Section IV: Working with Message Box Dialogs

Section V: Generating the Application Code

Note: The sections in this tutorial are independent of one another. If you are interested in learning about menus, for example, you can jump to Section II: Working with Menus, and start there (once you have started UIM/X and loaded the start-up project).

Section I: Getting Started and Drawing the Interface

In this section you will start UIM/X in Standard Mode and load the start-up project. Next you will create the color-changing Push Buttons, change a few properties, and add callback behavior to change properties at runtime. Finally, you will test the color-changing portion of the interface before proceeding to the next section.

The Steps in This Section

This section takes about 20 minutes to complete. It contains the following steps:

- Step #1: Starting UIM/X in Standard Mode
- Step #2: Loading the Start-Up Project
- Step #3: Laying Out the Working Area
- Step #4: Changing Labels and Other Properties
- Step #5: Adding Behavior to the Push Buttons
- Step #6: Testing the Color-Changing Push Buttons

Where You Are in the Tutorial

- ⇒Section I: Getting Started and Drawing the Interface
- Section II: Working with Menus
- Section III: Adding Line-Drawing Functionality
- Section IV: Working with Message Box Dialogs
- Section V: Generating the Application Code

Step #1: Starting UIM/X in Standard Mode

Before you begin building the Drawing Editor, set up a new directory and copy the start-up project into it as follows:

1. Start the X Window System.
2. Bring up a terminal window.
3. Make a directory to store the files you will create in this tutorial:

```
mkdir chap3
```
4. Change to the directory you just created:

```
cd chap3
```
5. Copy the required Drawing Editor project files into your work directory:

```
cp $UIMXDIR/contrib/MotifMain/* .  
cp $UIMXDIR/contrib/DrawDemo/graphics.c .  
cp $UIMXDIR/contrib/DrawDemo/*.xpm .
```
6. Change the permissions on the project files you copied to make them writable:

```
chmod a+w *
```

7. Start UIM/X from your new directory:

```
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 &
```

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and the Palette appear.

8. Iconify the terminal window.

Note: To restart the tutorial, begin again from Step 7 above.

Step #2: Loading the Start-Up Project

To facilitate development of the Drawing Editor, a start-up project has been provided. It contains the Drawing Editor primary interface with menus already defined, plus a Message Box and a file selection box. It also contains some source code in a separate file for creating application window behavior.

To load the start-up project:

1. Choose File⇒Open in the Project Window, or click on the Open icon in the Tool Bar.
2. Select `draw_start.prg`.

3. Select OK. The Drawing Editor start-up interface appears, as shown in Figure 3-2.

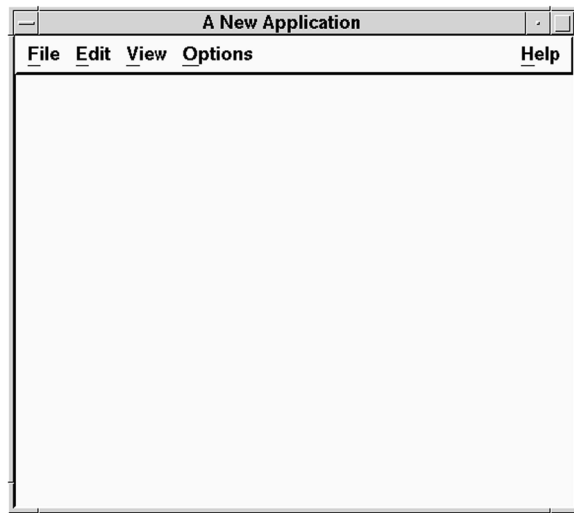


Figure 3-2 Drawing Editor Start-Up Project

The Message Box and File Selection Box provided are not visible by default, though icons for them appear in the Project Window. (To display a hidden interface double-click on its icon in the Project Window.) These interfaces are popped up by callbacks provided in the menus. You will test them later in Section II: Working with Menus.

Step #3: Laying Out the Working Area

In this step you will create the Drawing Editor's working area—a Scrolled Window containing a Frame. Next you will add the Push Buttons for changing the background color of the scrollable Frame.

Drawing the Scrolled Window and Frame

In this step you will add a Scrolled Window to the Drawing Editor by dragging and drawing. Then you will drag and drop a Frame into it, resizing it to make it larger than the Scrolled Window (making the scroll bars visible).

1. Make sure you are in Design Mode. If not, click on the Design icon in the Project Window.

- In the Managers area of the Palette, click on the Scrolled Window icon with the Select mouse button (the left one), as shown in Figure 3-3.

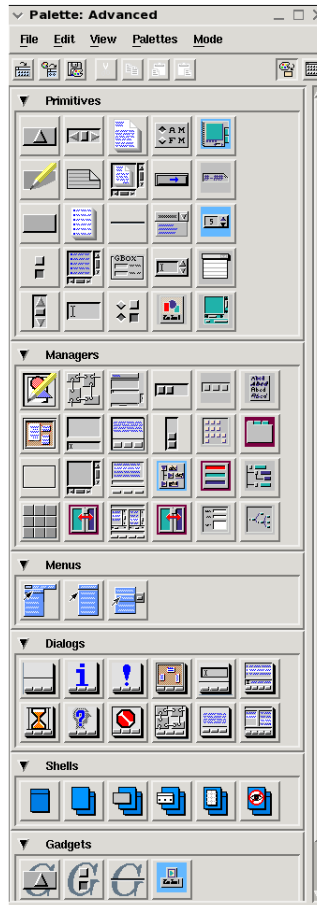


Figure 3-3 Selecting a Scrolled Window from the Palette

- Move the mouse pointer to where you want the upper-left corner of the Scrolled Window to appear. Use Figure 3-4 as a guide.
- Press and hold the Select mouse button, then drag the mouse downwards and to the right to draw the Scrolled Window. To complete the operation, release the mouse button.

The Scrolled Window appears as shown in Figure 3-4. Note that no scroll bars are displayed. In order for scroll bars to appear, the Scrolled Window must contain an object larger than its display area.

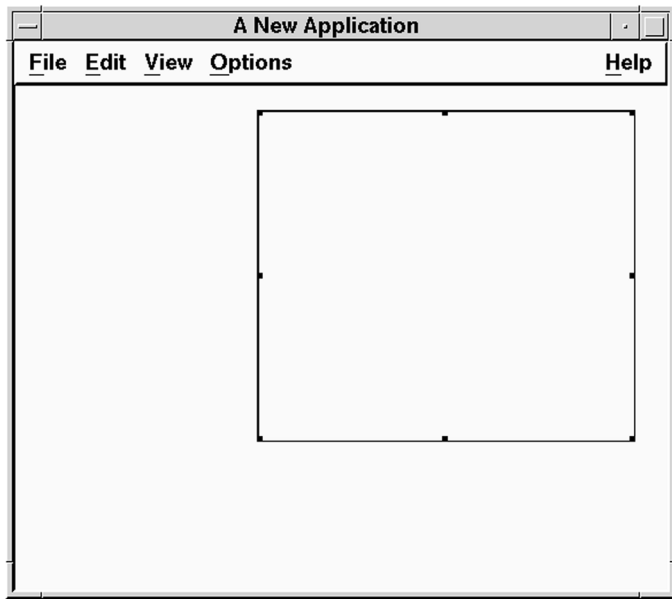


Figure 3-4 Drawing Editor with Scrolled Window added

5. To add a Frame to the Scrolled Window by dragging and dropping, begin by pointing to the Frame icon in the Managers area of the Palette.
6. Press and hold the Adjust mouse button (the middle one).
If you press the Select mouse button by mistake, press Escape to cancel the operation. Most mouse operations can be cancelled in this way.
The pointer changes to the compass shape, and an outline of the widget appears. This means the widget is ready for you to drag and drop it.
7. Drag the outline of the widget onto the main window, and release it over the Scrolled Window.
8. The Frame appears in its default size in the upper left corner of the Scrolled Window, as shown in Figure 3-5.

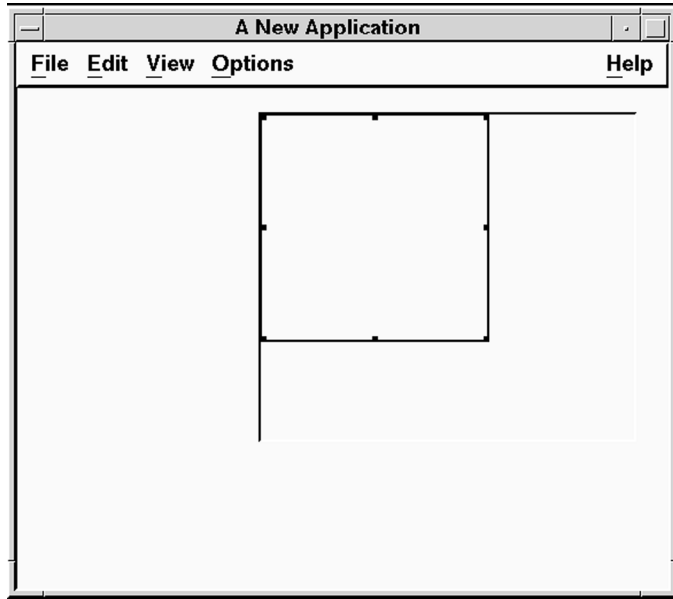


Figure 3-5 Drawing Editor with Frame Added to Scrolled Window

9. Resize the Frame until it is larger than the Scrolled Window using the resize grid, as shown in Figure 3-6.

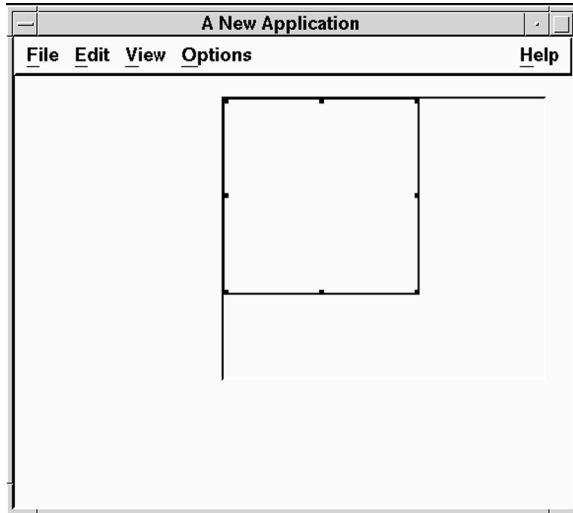


Figure 3-6 Using the Resize Grid to Resize the Frame

Note that scroll bars appear when you release the mouse, as shown in Figure 3-7.

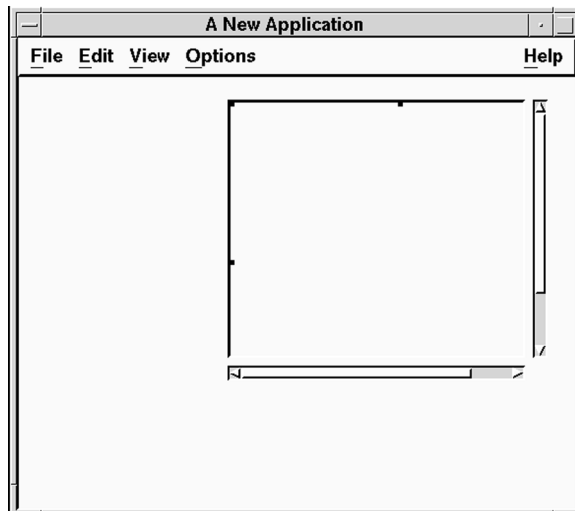


Figure 3-7 Scrolled Window Showing Scroll Bars

Note: If you need to move the Scrolled Window, select it by clicking on a scroll bar, then choose Selected Objects⇒Other⇒Move. If you try to move it using the resize grid alone, you will move the Frame it contains instead.

10. Save your work as a new project, `DrawingEditor.prj`, by selecting File⇒Save Project As.

Adding the Color-Changing Push Buttons

The Drawing Editor features four Push Buttons used to change the background color of the drawing area (the Frame in the Scrolled Window). In this step you will create a Push Button and duplicate it.

1. In the Primitives category of the Palette, click on the Push Button icon.
2. Drag and draw the Push Button, aligning it with the left edge of the Scrolled Window, using Figure 3-8 as a model.

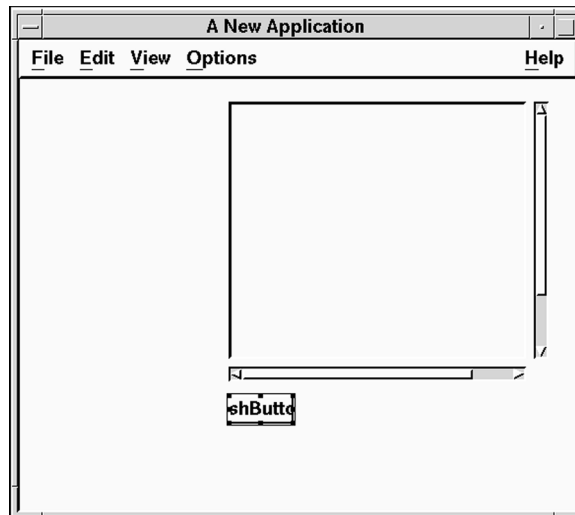


Figure 3-8 Primary Interface with First Push Button

Don't worry if the Push Button is not large enough to contain its label. You will remove the label later, when changing properties.

3. Duplicate the Push Button by choosing Selected Objects⇒Duplicate.
4. Create the final two Push Buttons (for a total of four) by duplication.
5. Align and distribute the Push Buttons, using Figure 3-9 as a model.

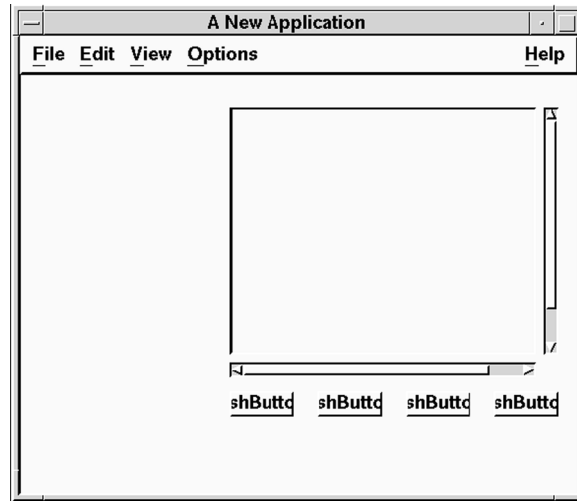


Figure 3-9 Primary Interface with All Four Push Buttons Added

6. Save your work.

Step #4: Changing Labels and Other Properties

Now that the drawing area and color-changing Push Buttons are in place on the interface, you are ready to change their labels and other properties. In this step you will begin by removing the Push Buttons' default labels. Next, you will change their background colors using the Color Editor.

Removing the Push Buttons' Default Labels

UIM/X features the ability to edit the properties of several widgets at once. In this step you will load the Push Buttons into the Property Editor together, and remove their labels by setting all four Push Buttons' `LabelString` properties.

1. Select all four Push Buttons by marquee selection, or by Ctrl-clicking. Press and hold the Select mouse button then drag the marquee around the Push Buttons, or hold down the Control key and click on each widget in turn.
2. Press the Menu mouse button, and choose Selected Objects⇒Tools⇒Property Editor.

- The Property Editor appears, loaded with the Push Buttons, as shown in Figure 3-10.

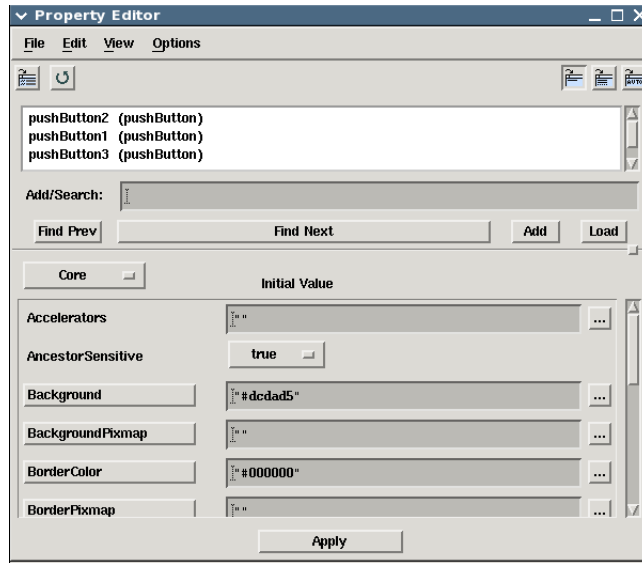


Figure 3-10 Property Editor Loaded with the Push Buttons

- In the Specific category of properties, locate the `LabelString` property, changing it to the empty string ("").
 Note that the initial value appears to be blank, and a “not-equals” icon appears beside the property. When more than one widget is loaded into the Property Editor, the “not-equals” icon indicates that the same property has a different value in at least one of the widgets.
- Apply the changes by clicking on `Apply` in the Property Editor.
 The interface is updated to reflect the changes, as shown in Figure 3-11.

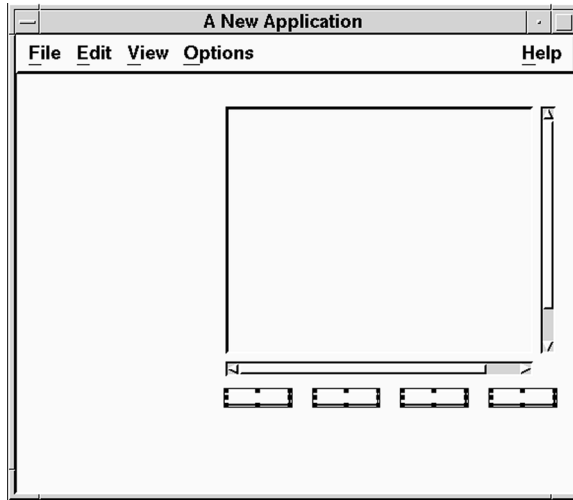


Figure 3-11 Drawing Editor with Empty Strings on Push Buttons

6. Save your work.

Changing the Push Buttons' Background Colors

In the final step in laying out the color-changing Push Buttons, you will change their background colors. To select a color you will use the Color Viewer, which gives access to your system's color database. You can optionally mix a custom color using UIM/X's Color Editor. The colors you assign the Push Buttons will be used later, to set the background color of the drawing area (the Frame in the Scrolled Window).

1. Load the first Push Button into the Property Editor by selecting it individually and dragging and dropping.
Since you will give each Push Button a different color, you cannot change them all at once.
2. In the Core set of properties, click on the Background property Push Button to open the Color Viewer.

The Color Viewer appears, as shown in Figure 3-12.

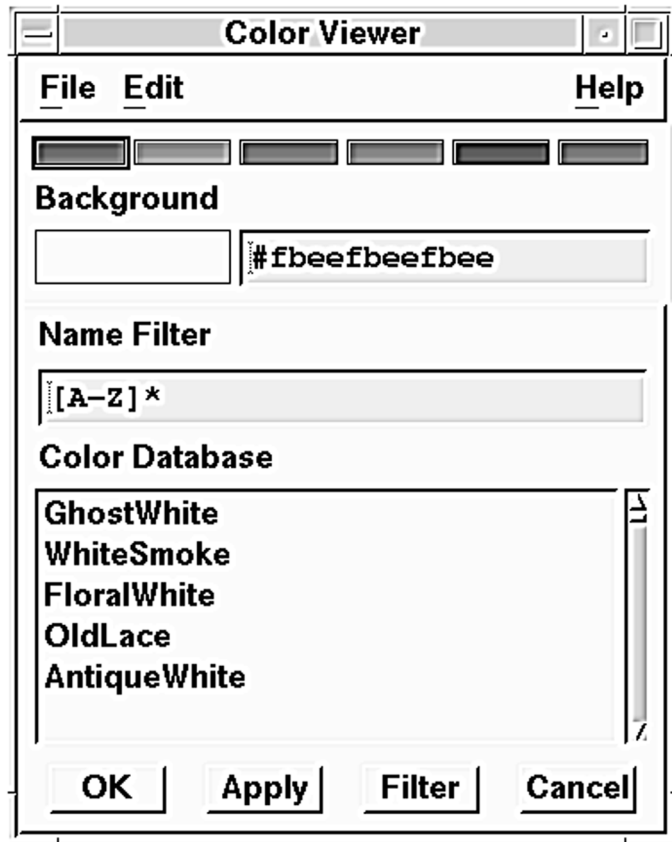


Figure 3-12 Color Viewer

3. Choose a background color for the Push Button in one of three ways:
 - Select a color from the Palette across the top of the Color Viewer.
 - Scroll through the Color Database and select a color by name.
 - Type an RGB value into the Background field.

Note: The color selections that you make should contrast with the foreground color (by default, "black") so that lines, shapes, and text drawable later in this tutorial will be visible.

4. Click OK to apply your choice to the Property Editor and close the Color Viewer, or Apply to apply your choice without closing it.

5. To create a custom color, open the Color Editor by selecting Edit⇒Edit Color from the Color Viewer menu bar.

The Color Editor appears as shown in Figure 3-13.

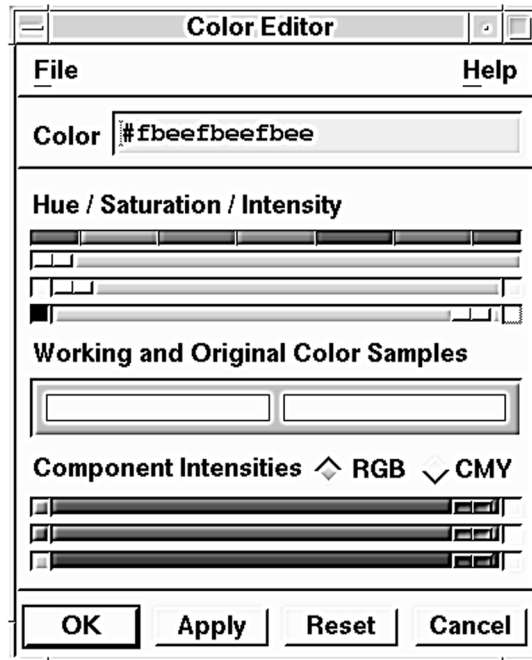


Figure 3-13 Color Editor

6. Create a custom color using the sliders:
 - Use the sliders to edit the Hue-Saturation-Intensity (HSI), Red-Green-Blue (RGB), or Cyan-Magenta-Yellow (CMY) color balance.
 - The colors at the end of each slider show the color obtained by moving the slider all the way to that end.
 - The working color on the left changes as you mix the new color.
7. Once you are satisfied with your color, click OK in the Color Editor. The color is copied to the Background area of the Color Viewer.
8. Click OK in the Color Viewer to apply the color to the Property Editor. The hexadecimal value for the color is displayed in the Background property. For colors chosen from the Color Database, the name is displayed.

9. Apply your change to the Push Button by clicking Apply in the Property Editor.
10. Repeat the above steps for the remaining Push Buttons. Select from the Color Database using the Color Viewer, or compose new colors using the Color Editor. Don't forget to apply your changes at each step. For the final Push Button, close the Color Editor and Color Viewer by clicking on OK.
11. Save your work.

Step #5: Adding Behavior to the Push Buttons

Now that you have laid out the working area of the interface, changed captions and background colors, the next step is to add behavior to the Push Buttons. While UIM/X components contain a great deal of built-in behavior—clicking on a Push Button changes its graphical representation, for example—advanced behavior must be added by writing callback code.

Callback code is automatically executed when the user triggers its corresponding event. A Push Button's `ActivateCallback`, for example, is triggered when the user clicks on the Push Button. Other widgets contain callbacks particular to their special uses.

Since each color-changing Push Button performs the same task using the same callback code, in this step you will load all four Push Buttons into the Property Editor at once. You will then use a Ux Convenience Library function, `UxPutBackground`, to set the background color of the Frame.

To add behavior to the Push Buttons:

1. Select all four Push Buttons and load them into the Property Editor.
2. Open the Callback Editor by clicking on the Push Button [...] beside `ActivateCallback` (in the Behavior category).

The Callback Editor appears as shown in Figure 3-14.

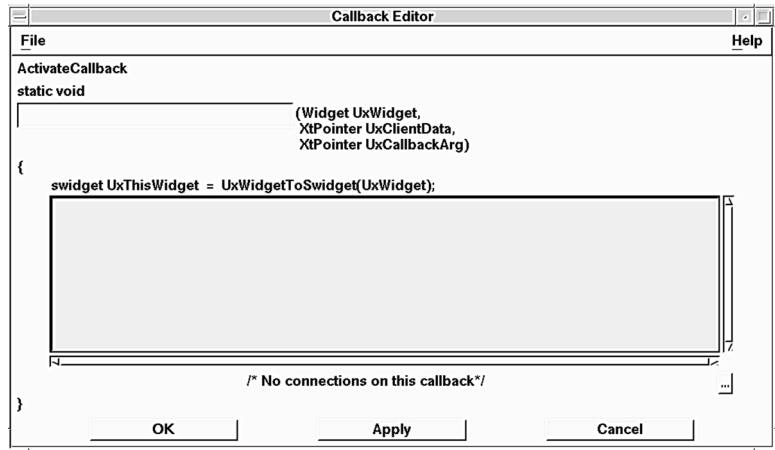


Figure 3-14 Callback Editor

3. Click in the Callback Editor Text Field, and type the following callback code:


```
UxPutBackground ( frame1 , UxGetBackground ( UxThisWidget ) ) ;
```
4. Click on OK on the Callback Editor to update the Property Editor with your entry.
5. Click on Apply in the Property Editor to save your changes and update all four Push Buttons at once.
6. Close the Property Editor by selecting File⇒Close from the Property Editor menu.
7. Save your work. You are now ready to test this portion of the interface.

Step #6: Testing the Color-Changing Push Buttons

Before beginning to work with the menus in the next part of this tutorial, take a moment to switch to Test Mode. Test Mode allows you to see how your interface will behave at runtime, without the need to compile code or exit the development environment.

1. Switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

2. Test the color-changing Push Buttons:
 - Clicking on a Push Button changes the color of the Frame.
 - Use the scroll bars in the Scrolled Window.
3. When you are through, switch back to Design Mode by clicking on the Design icon

Section II: Working with Menus

In UIM/X working with menus is simplified for two main reasons. First, menu elements contain built-in behavior including automatic sizing and positioning. You never have to worry about the size of menu labels, or pull-down behavior, for example. Second, UIM/X features a Menu Editor that provides a structured means to build your menu bar and add items to the menus.

In this section you will work with the menu bar already provided with the start-up project, adding a pulldown menu to it. You will also add a cascading menu, illustrating how to create an additional level of structured access to your application's commands. At runtime choosing an item in the menu will change the background color of the Frame. You will then test the menus.

The Steps in This Section

This section takes about 30 minutes to complete. It contains the following steps:

- Step #7: Adding a Pulldown Menu
- Step #8: Adding a Cascading Menu
- Step #9: Adding Behavior to the Color Menu
- Step #10: Testing the Menus

Where You Are in the Tutorial

- Section I: Getting Started and Drawing the Interface
- ⇒ Section II: Working with Menus
- Section III: Adding Line-Drawing Functionality
- Section IV: Working with Message Box Dialogs
- Section V: Generating the Application Code

Step #7: Adding a Pulldown Menu

In this step you will add a new pulldown menu to the menu bar provided, and populate it with items. At runtime choosing an item from the new Color menu will change the background color of the Frame.

To Add a Pulldown Menu:

1. Select any widget in the menu bar, and open the Menu Editor by choosing Selected Objects⇒Tools⇒Menu Editor.

The Menu Editor appears as shown in Figure 3-15.

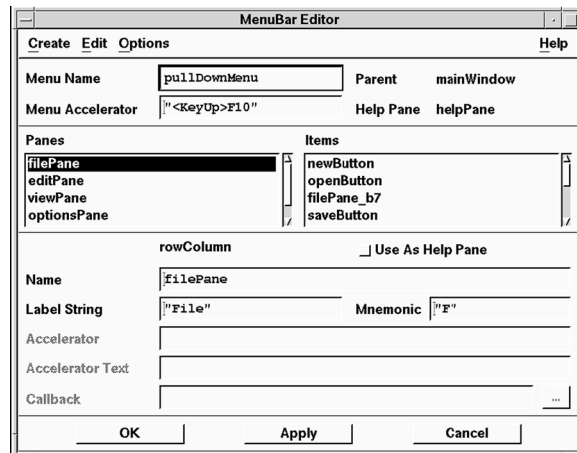


Figure 3-15 Menu Editor

2. Add a new pane by choosing Create⇒Pane from the Menu Editor.
3. Enter the values shown in Table 3-1 for the new pane, pullDownMenu_p6:

Table 3-1 Property Values for Color Menu Pane

Property	Value
Name	colorPane
Label String	"Color"
Mnemonic	"C"

4. Add an item to the new pane by selecting Create⇒Item After⇒Push Button.
5. Enter the values shown in Table 3-2 for the new Push Button:

Table 3-2 Property Values for White Menu Item

Property	Value
Name	colorWhite
LabelString	"White"
Mnemonic	"W"

6. Add a second Push Button with the values shown in Table 3-3:

Table 3-3 Property Values for Green Menu Item

Property	Value
Name	colorGreen
LabelString	"Green"
Mnemonic	"G"

7. Add a third Push Button with the values shown in Table 3-4:

Table 3-4 Property Values for Blue Menu Item

Property	Value
Name	colorBlue
LabelString	"Blue"
Mnemonic	"B"

8. Add a fourth (and final) Push Button with the values shown in Table 3-5:

Table 3-5 Property Values for Hot Pink Menu Item

Property	Value
Name	colorHotPink
LabelString	"Hot Pink"
Mnemonic	"P"

9. Click on Apply to apply your changes.

Note the new menu, Color, is added to the menu bar, as shown in Figure 3-16. The mnemonic you specified for the Color menu, "C", is underlined.

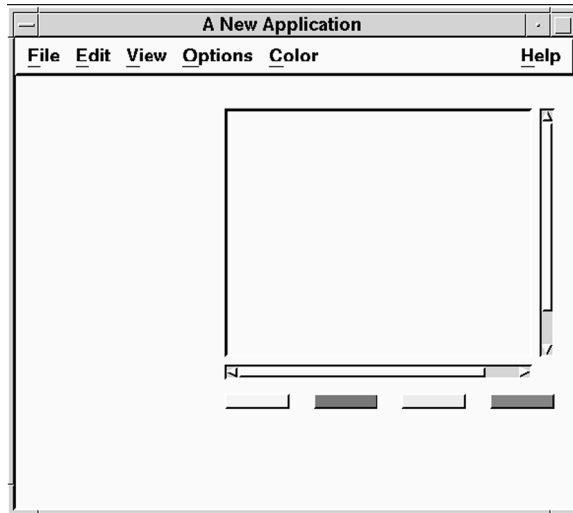


Figure 3-16 Drawing Editor with Color Menu Added

10. Save your work, leaving the Menu Editor open for the next step, adding a cascading menu.

Step #8: Adding a Cascading Menu

Cascading menus are a convenient way to provide choices without taking up too much room. When the user clicks on a pane designated as a cascading menu (indicated by a right-arrow), the submenu is presented. As with other elements of a menu, UIM/X takes care of positioning the cascading menu, providing the right-arrow graphics, and the cascading behavior itself.

In this step you will add a cascading menu to the Color menu just created. The cascading menu will be called Grayscale, and will contain three items: Light Gray, Medium Gray, and Dark Gray. First you will create the submenu, then you will add the cascade button and connect it to the submenu. All work will be performed in the Menu Editor.

To add a cascading menu:

1. Create the new submenu by choosing Create⇒Pane from the Menu Editor.
A new pane, pullDownMenu_p7, is added to the end of the Panes list.
2. Add an item to the submenu by choosing Create⇒Item After⇒Push Button in the Menu Editor.

3. Enter the values shown in Table 3-6 for the new Push Button:

Table 3-6 Property Values for Light Gray Menu Item

Property	Value
Name	lightGray
LabelString	"Light Gray"
Mnemonic	"L"

4. Add a second Push Button with the values shown in Table 3-7:

Table 3-7 Property Values for Medium Gray Menu Item

Property	Value
Name	mediumGray
LabelString	"Medium Gray"
Mnemonic	"M"

5. Add the third and final Push Button with the values shown in Table 3-8:

Table 3-8 Property Values for Dark Gray Menu Item

Property	Value
Name	darkGray
LabelString	"Dark Gray"
Mnemonic	"D"

6. To create the Cascading menu, begin by selecting the Color menu by scrolling through the list of panes in the Menu Editor, and selecting colorPane.

The list of items contained in the Color menu appear in the Items area, as shown in Figure 3-17.

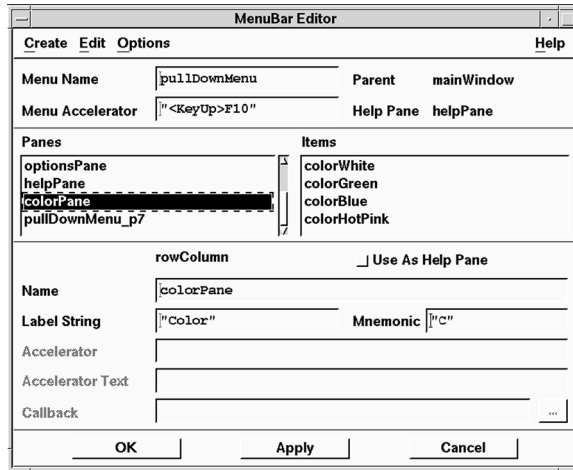


Figure 3-17 Menu Editor Showing colorPane's Items

7. In the Items list, select colorHotPink.
8. Add a cascading menu after the Hot Pink item by choosing Create⇒Item After⇒Cascade Button in the Menu Editor.

A new item, colorPane_b5, is added to the end of the Items list.

9. Enter the values shown in Table 3-9 for the cascading menu:

Table 3-9 Property Values for Cascade Menu Pane

Property	Value
Name	grayCascade
LabelString	"Grayscale"
Mnemonic	"g"
Next Pane	pullDownMenu_p7

The value for Next Pane links the Cascade Button to the pulldown menu you created earlier.

10. Click on OK to apply your changes and close the Menu Editor.
11. Save your work.

Step #9: Adding Behavior to the Color Menu

To simplify connecting interface elements together, UIM/X features a Connection Editor. By loading both the source and target widgets into the editor, you can view the available callbacks in the source, and the methods in the target. You can then *connect* the source's callback to the target's method visually, rather than via callback code.

In this step you will load the Color menu's items into the Connection Editor, connecting their `ActivateCallback` callbacks to the Frame's `SetBackground` method. First you will open the editor and make the first connection. Next you will load the remaining menu items into the editor, one by one, and connect them. Since menu items are not visible in the interface at design time, you will use the Browser to view and select the items for loading.

Opening the Connection Editor and Making the First Connection

In this step you will load the Color menu's White item into the Connection Editor and create a connection to change the Frame's background color.

1. Open the Browser by choosing Selected Objects⇒Tools⇒Browser while over the Main Interface.

Since menus contain many widgets, you might find it convenient to view the widgets by Name only.

2. In the Browser click on the `colorWhite` menu item (under `colorPane`), as shown in Figure 3-18.

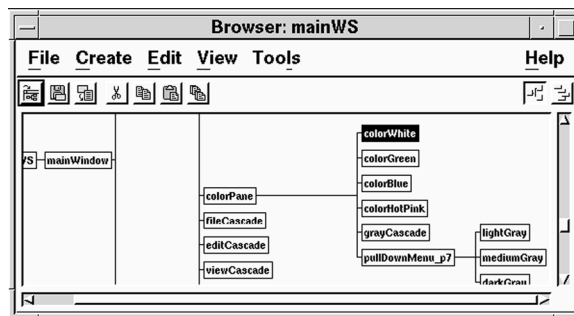


Figure 3-18 Browser Showing `colorWhite` Menu Item Selected

3. Open the Connection Editor by choosing Selected Objects⇒Tools⇒Connection Editor while over the Browser.

The Connection Editor appears loaded with `colorWhite` item in the Source area, as shown in Figure 3-19. Notice `colorWhite`'s callbacks are listed in the Callback area of the Connection Editor.

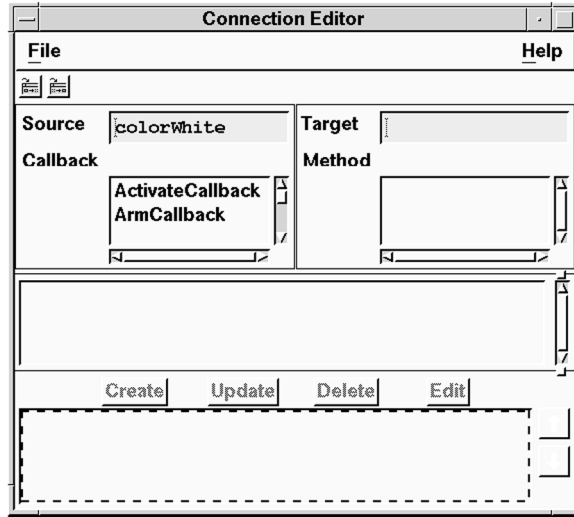


Figure 3-19 Connection Editor Showing `colorWhite`'s Callbacks

4. Load the Frame into the Target area of the Connection Editor in one of two ways:
 - Click on it with the Adjust mouse button, then drag and drop it into the Target area of the Connection Editor.
 - Click on it with the Select mouse button and click on the Load Target icon (the right-most one) in the Connection Editor.

The instance's default methods are listed in the Method area of the Connection Editor, as shown in Figure 3-20.

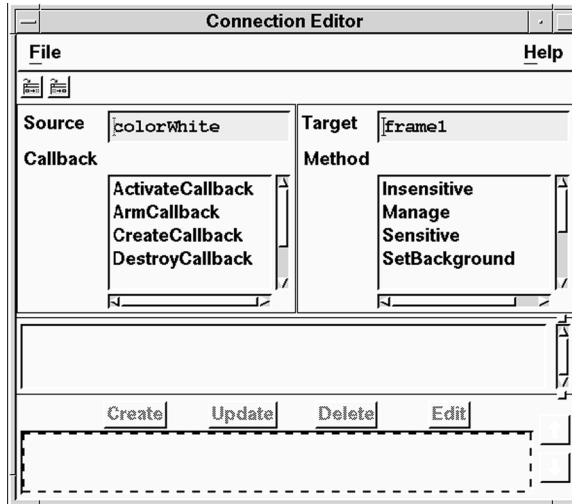


Figure 3-20 Connection Editor Showing frame1's Methods

5. Click on `ActivateCallback` in list of callbacks, and on `SetBackground` in the list of methods.

The `Color` parameter appears in the parameters area.

6. Replace the default value, "black", with the desired value, "white".
7. Complete the connection by clicking on `Create`.

If your system's color database does not contain a definition for "white", you will receive an error message. Edit the connection, substituting the color white's hexadecimal value, "#fafafafafafa", instead. Be sure to enclose the hexadecimal value in quotation marks.

To see the colors for which strings are defined, use UIM/X's Color Viewer.

8. The new connection appears in the Connection Editor, as shown in Figure 3-21.

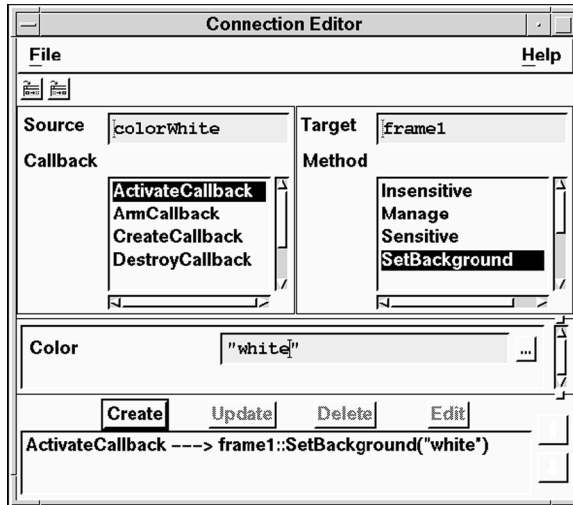


Figure 3-21 Connection Editor Showing New Connection

9. Save your work.

Making the Remaining Connections

In this step you will create the remaining Color menu connections, selecting menu items in the Browser and loading them into the already open Connection Editor.

1. Select the `colorGreen` menu item in the Browser.
2. Load it into the Source area of the Connection Editor by clicking on the Load Source icon (the left one) or choosing `File⇒Load Source` in the Connection Editor.

Note that the Color parameter retains the previous setting, for convenience.

3. Replace `"white"` with `"green"` then complete the connection by clicking on Create.

As before, if `"green"` is not defined on your system, you will have to enter the hexadecimal value instead (or an equivalent string).

4. Repeat the process for the remaining menu items. Table 3-10 list all the menu items and the values you should assign the `Color` parameter. For convenience the table lists the hexadecimal values as well.

Table 3-10 Values for Color Parameter for All Menu Items

Menu Item	Value for Color	Hexadecimal Value
colorWhite	"white"	"#fafafafafafa"
colorGreen	"green"	"#0000ffff0000"
colorBlue	"blue"	"#51005100fb00"
colorHotPink	"hot pink"	"#ffff6969b4b4"
lightGray	"light gray"	"#d3d3d3d3d3d3"
mediumGray	"gray"	"#bebebebebebe"
darkGray	"dark slate gray"	"#2f2f4f4f4f4f"

5. When complete close the Connection Editor by choosing File⇒Close.
6. Save your work.


Step #10: Testing the Menus

Before adding the drawing functionality in the next section, take a moment to test the menus.

1. Switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

2. Test the pulldown behavior:
 - To display a pulldown menu, click on the menu bar.
 - Dragging the cursor highlights the menu items.
3. Test the menu you added:
 - Change the drawing area's color by choosing an item from the Color menu.
 - Choose Color⇒Grayscale to display the cascading menu.

4. Test the functionality provided with the start-up project:
 - File⇒Open and File⇒Save As pop up the File Selection Box provided with the project. Selecting a file prints a message to the Project Window message area.
 - File⇒Exit pops up the Message Box provided. UIM/X traps any exit command, printing a message instead.
 - Selecting an item on the Edit menu prints a corresponding message to the message area.
 - Items on the View menu change the background color of the Main Window, printing a message in the message area.
 - The Options menu contains multiply-selectable options (the *general* options) and mutually exclusive options (the *radio* options).
5. Test the keyboard control:
 - To open a menu using the keyboard, press Alt and the menu's mnemonic (the underlined letter).
 - Use the arrow keys to move between menus and menu items, or simply press the next mnemonic.
 - To choose a menu item without opening the menu use its keyboard accelerator. For example, pressing Shift-Del prints "Cut !" to the message area.
6. When you are through, switch back to Design Mode by clicking on the Design icon 
You are now ready to add the line-drawing functionality, in the next section.

Section III: Adding Line-Drawing Functionality

In UIM/X widgets are provided with a great deal of built-in behavior, and more complex behavior is easily added via callbacks. To respond to events such as mouse clicks and mouse motion, a more general technique is required. For example, dragging the cursor through the working area of a Text Editor application would most likely select text. In a Drawing Editor application, you would expect different behavior. This third kind of response is most often provided by the underlying application, rather than the interface. In UIM/X this is referred to as application window behavior.

Specifying application window behavior is done via translation tables, structures that link application window events to application actions. Using the Translation Table List and its associated editors, you can graphically specify the mouse and keyboard events to which you want to respond, and create links to your application libraries.

The advantages of translation tables are two-fold. First, translation tables are shared by the project as a whole. This makes it easy for two separate portions of the interface to initiate the same kind of behavior. Second, you can easily activate and deactivate translation tables, substituting behavior as required. In this way the same mouse event can trigger different responses, depending on the state of your application.

In this section you will use translation tables to add line-drawing functionality to the Drawing Editor. You will begin by drawing the new Push Buttons. Next you will use the Translation Table Editor to create different application window behavior for each Push Button, linking mouse events to the library of graphics commands provided with the start-up project. You will then use the Push Button's `ActivateCallback` to activate the appropriate translation table. Finally, you will test the line-drawing functionality.

The Steps in This Section

This section takes about 30 minutes to complete. It contains the following steps:

- Step #11: Creating the Line-Drawing Push Buttons
- Step #12: Creating the Application Window Behavior
- Step #13: Applying the Behavior to the Line-Drawing Push Buttons
- Step #14: Testing the Line-Drawing Push Buttons

Where You Are in the Tutorial

- Section I: Getting Started and Drawing the Interface
- Section II: Working with Menus
- ⇒Section III: Adding Line-Drawing Functionality
- Section IV: Working with Message Box Dialogs
- Section V: Generating the Application Code

Step #11: Creating the Line-Drawing Push Buttons

In this step you will draw the Push Buttons, and use the Icon Viewer to preview and load bitmaps into each one.

To add the line-drawing Push Buttons:

1. Drag and drop a Push Button from the Primitives area of the Palette to your main window interface.

Position it to the left of the Scrolled Window, as shown in Figure 3-22.

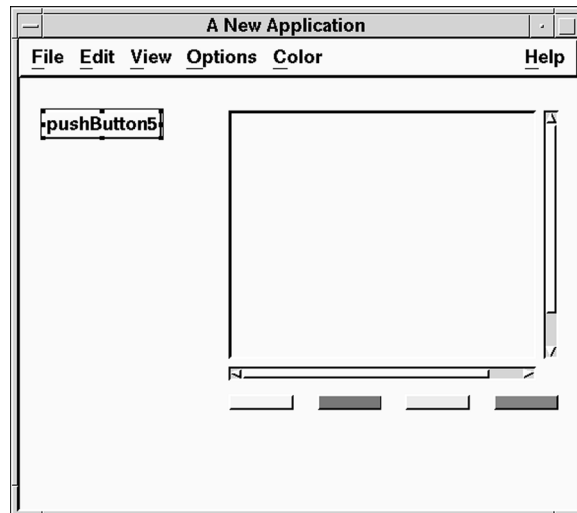


Figure 3-22 Drawing Editor with First Line-Drawing Push Button Added

2. Create three more Push Buttons by dragging and drawing, dragging and dropping, or duplication.

Position the new Push Buttons under the first one. Don't worry if they are not all the same size. They will resize automatically when you add their pixmaps.

3. To add a pixmap to the first Push Button, `pushButton5`, begin by loading it into the Property Editor.
Double click on it, or select it (by clicking once) and choose Selected Objects⇒Tools⇒Property Editor.
4. Locate the `LabelType` property in the Specific category, changing it from `string` to `pixmap`.
5. Locate the `LabelPixmap` property in the Specific category and open the Icon Viewer by clicking on the `LabelPixmap` button.

The Icon Viewer appears, as shown in Figure 3-23.

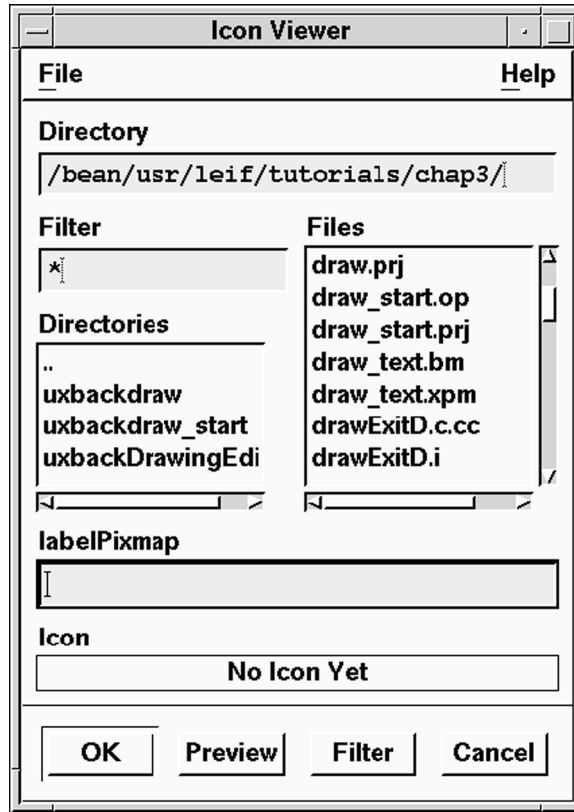


Figure 2-23 Icon Viewer

6. Find the line pixmap, `line.xpm`:
 - To list only the bitmaps in your current directory, enter `*.xpm` in the Filter area, then click on the Filter button.
 - To preview a bitmap, highlight the file name. It will appear automatically in the Pixmap area of the Icon Viewer.
7. Load the line icon into the Property Editor by clicking OK in the Icon Viewer.
8. Click on Apply in the Property Editor.

9. The line icon is loaded into the Push Button, which resizes to fit it. Your application should now look similar to Figure 3-24.

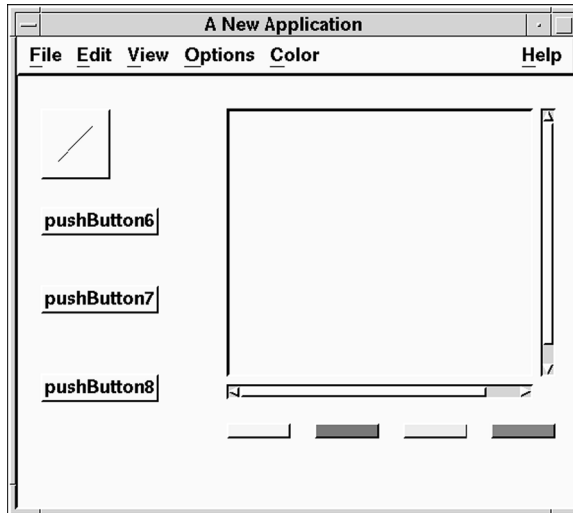


Figure 3-24 Line-Drawing Push Button, `pushButton5`, with Pixmap Added

10. Repeat the process, adding pixmaps to the remaining three Push Buttons.

Table 3-11 lists all the Push Buttons and the pixmaps they should contain:

Table 3-11 List of Pixmaps for the Push Buttons

Widget Name	Pixmap
<code>pushButton5</code>	<code>line.xpm</code>
<code>pushButton6</code>	<code>circle.xpm</code>
<code>pushButton7</code>	<code>rectangle.xpm</code>
<code>pushButton8</code>	<code>ellipse.xpm</code>

11. When you are done, close the Property Editor by choosing File⇒Close.

12. Align and arrange the line-drawing Push Buttons. When complete, the interface should look as shown in Figure 3-25.

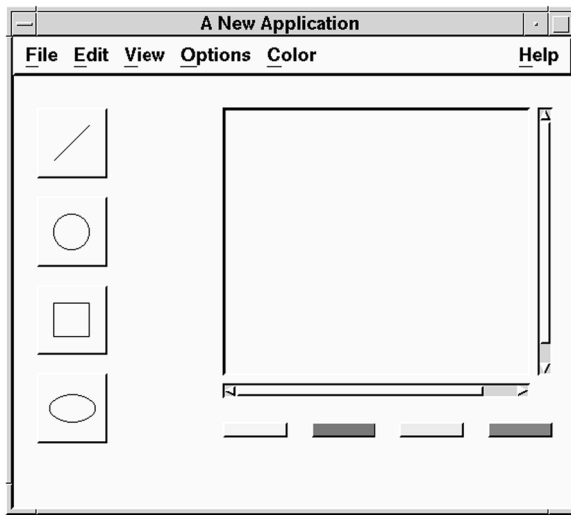


Figure 3-25 Drawing Editor with Pixmaps Added

13. Save your work.

Step #12: Creating the Application Window Behavior

UIM/X provides two editors to facilitate the specification of application window behavior: the Translation Table Editor and the Event Editor. It also provides a structured access to the translation tables in your project, via the Translation Table List.

The Translation Table List lets you pair user-generated events with application generated actions. You can also set the table policy to override, augment, or replace the current application window behavior.

While you can enter any X Toolkit event into the Translation Table Editor, you can also specify events graphically via the Event Editor. By simply pointing and clicking, you can define events for most mouse activity, as well as keyboard events.

In this step you begin by opening the editors associated with translation tables. Next you will initialize the interpreter with the graphics library provided with the start-up project. You will then create a translation table for each line-drawing Push Button.

Opening the Editors

In this step you will open the editors.

1. To begin, open the list of translation tables for the interface by choosing Selected Objects⇒Tools⇒Translation Table List.

The Translation Table List appears, as shown in Figure 3-26.

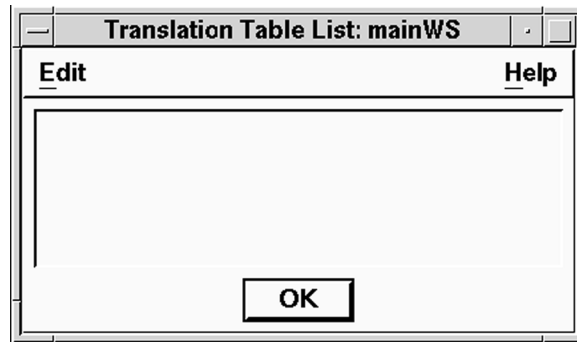


Figure 3-26 Translation Table List

2. Click <Select> the mainWS interface from the “Interfaces” area of the UIM/X Project window.
3. Add a translation table to the project and open the Translation Table Editor by choosing Edit⇒Add in the Translation Table List.

The Translation Table Editor appears, as shown in Figure 3-27.

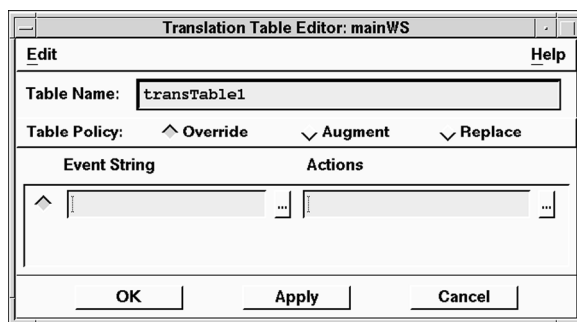


Figure 3-27 Translation Table Editor

4. Next, open the Event Editor by choosing Edit⇒Event Editor in the Translation Table Editor.

The Event Editor appears, as shown in Figure 3-28.

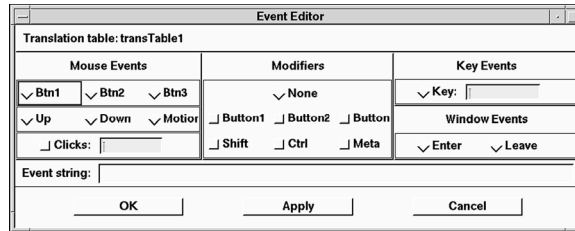


Figure 3-28 Event Editor

5. Position the three dialogs so you can work with them conveniently.

Initializing the Interpreter with the Action Code

Before specifying the actions that will occur in response to mouse events, you must initialize the interpreter with the action code. That way, when you specify responses in the translation table, UIM/X will accept the function calls without error.

1. Choose Tools⇒Interpreter in the Project Window. The interpreter appears, as shown in Figure 3-29.

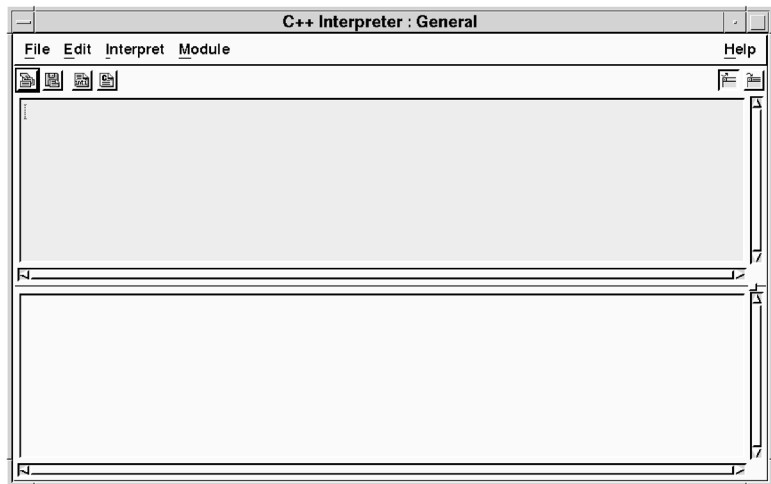


Figure 3-29 Interpreter

2. Choose File⇒Load Source Code in the interpreter.
 Loading source code into the interpreter makes the code module's functions available to the development environment. It is similar to compiling code for run-time execution.

- In the file selection box that appears, select `graphics.c` and click OK. The following message appears in the Messages Area of the Interpreter:

Result: OK

- Initialize the graphics code by typing the following line:

```
UxInitGraphics ();
```

`UxInitGraphics` is defined in `graphics.c`. It initializes the graphics code and registers the actions you will use later.

- To execute the function, double-click to highlight it and choose Interpret⇒Evaluate in the Interpreter, or click on the Evaluate icon.

The Messages Area now shows:

Result: 0

- Choose File⇒Close in the Interpreter.

Defining a Translation Table for the Line-Drawing Push Buttons

In this step you will define three events to match mouse clicks and motion in the application window. Using the Event Editor you will graphically define events, copying them to the translation table editor. Since the translation tables for the line, circle, rectangle, and ellipse Push Buttons are almost the same, you will then duplicate the translation table for the other Push Buttons.

- In the Translation Table Editor, replace the default table name, `transTable1`, with `Line`, and set the Table Policy to *replace*.
- In the Event Editor, click on the Btn1 and Down radio buttons in the Mouse Events area.

`<Btn1Down>` appears in the Event String area, as shown in Figure 3-30.

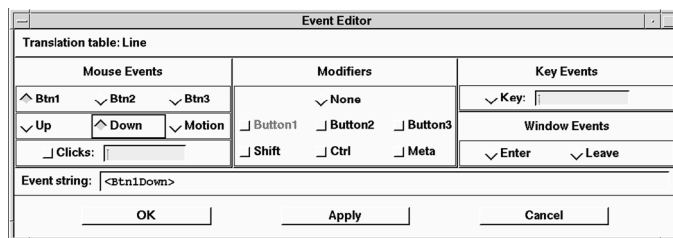


Figure 3-30 Event Editor Showing `<Btn1Down>` Event

- Copy the event to the Translation Table Editor by clicking on Apply.

The `<Btn1Down>` event appears in the Translation Table Editor, as shown in Figure 3-31.

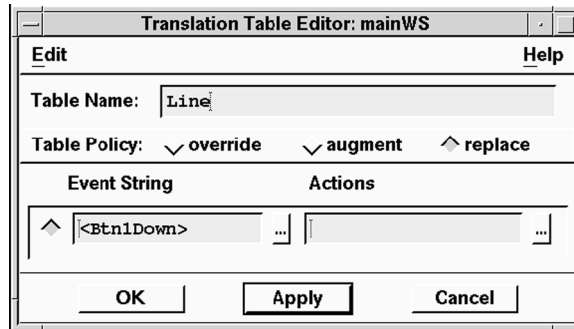


Figure 3-31 Translation Table Editor Showing `<Btn1Down>` Event

4. In the Actions area, add the following action:

```
first_point
```

The `first_point` function is defined in the `graphics.c` file loaded earlier.

5. Apply the change by clicking on Apply in the Translation Table Editor. UIM/X automatically adds any missing parentheses to the function call.
6. Add a new event-action pair by choosing Edit⇒Add in the Translation Table Editor.
An empty pair is added.
7. Repeat the process, creating new event strings in the Event Editor, copying them to the Translation Table Editor, and adding the appropriate actions. Table 3-12 lists all the event-action pairs required by the *Line* translation table.

Table 3-12 Event-Action Pairs for *Line* Translation Table

Events	Actions
<code><Btn1Down></code>	<code>first_point</code>
<code><Btn1Motion></code>	<code>draw_line</code>
<code><Btn1Up></code>	<code>last_point</code>

When complete the Translation Table Editor should appear as shown in Figure 3-32.

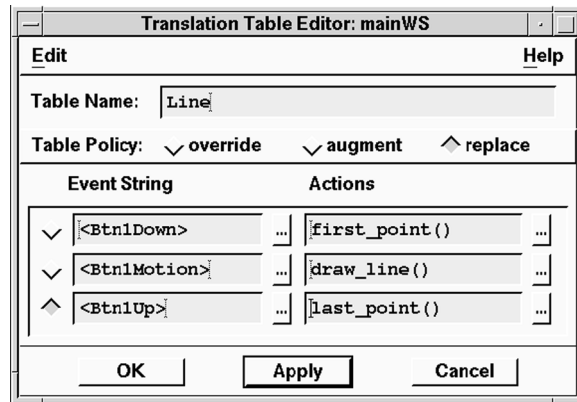


Figure 3-32 Translation Table Editor Showing All Events Needed

8. Click on OK to apply your changes and close the Translation Table Editor.

Creating Translation Tables for the Other Push Buttons

In this step you will duplicate the translation table just created for each of the remaining Push Buttons. You will change the action for the `<Btn1Motion>` event to a more appropriate function call.

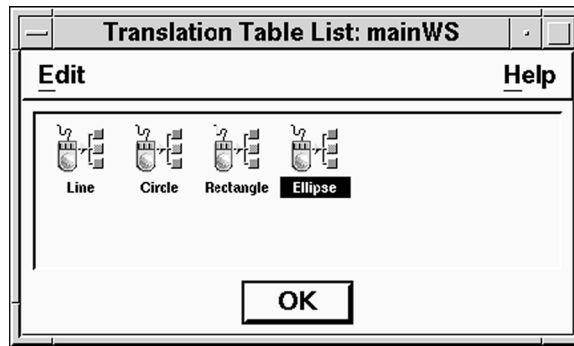
1. To duplicate the translation table, begin by selecting the *line* translation table by clicking on it in the Translation Table List.
2. Choose Edit⇒Duplicate in the Translation Table List.
A new translation table is created, `transTable1`, and its icon is added to the list.
3. Double-click on the new icon to open its Translation Table Editor.
4. Replace the existing Table Name `transTable1` with `Circle`.
5. Since the *circle* Push Button should draw a circle and not a line, replace the `<Btn1Motion>` action, `draw_line`, with a more appropriate action, `draw_circle`.
6. Apply your changes and close the editor by clicking on OK in the Translation Table Editor.
7. Repeat the process, creating two more translation tables, for the rectangle-drawing and ellipse-drawing Push Buttons respectively.

Table 3-13 lists the translation tables for all four Push Buttons, with the appropriate actions for the `<Btn1Motion>` events:

Table 3-13 <Btn1Motion> Actions for the Four Translation Tables

Push Button Name	Translation Table	<Btn1Motion> Action
pushButton5	Line	draw_line
pushButton6	Circle	draw_circle
pushButton7	Rectangle	draw_rectangle
pushButton8	Ellipse	draw_ellipse

After your entries, the Translation Table List should now look as shown in Figure 3-33.

*Figure 3-33* Translation Table List with All Four Icons

8. Click on OK to close the Translation Table List.
9. Save your work.

Step #13: Applying the Behavior to the Line-Drawing Push Buttons

With the translation tables created, it now remains to apply each one to the Frame in the Scrolled Window at the appropriate moment. In UIM/X you can attach a translation table to an object at design time using the `Translations` property, or dynamically at runtime using `UxPutTranslations`.

In this step you will add behavior to each of the Push Buttons to apply its translation table to the Frame inside the scrolled window.

1. Double-click on the line-drawing Push Button, `pushButton5` to open the Property Editor.

2. In the Behavior category, locate the `ActivateCallbackevent`, and type the following callback code into it:

```
UxPutTranslations (frame1, Line) ;
```

1. Apply the change by clicking on `Apply` in the Property Editor.
1. Repeat the process for each of the remaining Push Buttons, substituting the appropriate translation tables.

Table 3-14 lists the `ActivateCallback` code for all four Push Buttons:

Table 3-14 ActivateCallback Code for the Four Push Buttons

Push Button Name	TranslationTable	ActivateCallback Codes
pushButton5	Line	<code>UxPutTranslations (frame1, Line) ;</code>
pushButton6	Circle	<code>UxPutTranslations (frame1, Circle) ;</code>
pushButton7	Rectangle	<code>UxPutTranslations (frame1, Rectangle) ;</code>
pushButton8	Ellipse	<code>UxPutTranslations (frame1, Ellipse) ;</code>

2. Close the Property Editor by selecting `File⇒Close` from the Property Editor menu.
3. Save your work. You are now ready to test this portion of the interface.

Step #14: Testing the Line-Drawing Push Buttons

Before continuing with the tutorial, take a moment to test the work you have done in this section.

1. Switch to Test Mode by clicking on the Test icon in the Project Window

The Palette and any other open editors disappear. The UIM/X main window and your interface remain.

2. Test the line-drawing functions:
 - Draw a line, circle, rectangle, or ellipse by clicking on the appropriate Push Button, then dragging and drawing in the Scrolled Window.
 - If it is difficult to see the drawn object, change the background color of the frame to increase the contrast.

3. When you are through, switch back to Design Mode by clicking on the Design icon

Section IV: Working with Message Box Dialogs

UIM/X features a number of dialog widgets designed to convey information to users, and simplifies working with dialogs in several ways. By using an instance of a dialog in your calling interface, for example, you protect it from unwanted changes. In addition, you can create property accessor methods to greatly simplify reading or writing a custom message to the dialog. Finally, the instance and accessor method combination simplify popping up the interface.

Some dialog widgets are well-suited to displaying simple messages, others to asking a question and obtaining a *yes* or *no* answer. Still others contain graphics conveying the degree of urgency of the message. In UIM/X all dialogs share a convenience of use, and the ease with you can display custom messages dynamically.

Placing an *instance* of the dialog in the interface where you will call it protects the dialog from modification. As with other widgets, creating an instance of it renders most of its properties unavailable in the instance. This is ideal for distributing a modified dialog throughout your design team, or simply for maintaining a consistent look when the dialog is used in different interfaces. Changes to the original dialog are of course possible, and are automatically reflected in all the instances.

To make the message area—or any property—available for reading or writing in the instance you create property accessor methods for the original dialog. Property accessor methods are pairs of methods following a specific naming convention: `ObjectName_get_MethodName` and `ObjectName_set_MethodName`. You provide the method names, while UIM/X provides the prefixes. In the body of the method, you “expose” the property using the `UxGetProperty` and `UxSetProperty` functions.

When UIM/X identifies a pair of *get* and *set* accessor methods in an instance, it presents a *MethodName* property in the Property Editor. Setting the new property calls the underlying method for the instance. Since the new property behaves like any other, you can provide it with a default value using the Property Editor, or set it at run time in callback code. You can also make a connection to it using the Connection Editor.

There is a final advantage to using instances and property accessor methods for dialogs. The property accessor methods and the properties they make available become local to the calling interface. Therefore, there is no need to declare global variables for the dialog, or the interface from which it is called.

In this section you will add a Message Box dialog to the Drawing Editor interface. To facilitate displaying a message, you will create an interface method for the dialog, and add an instance of the dialog to the main interface. Next you will use the Connection Editor to pop up the dialog from the Push Button, and write the contents of the Text Field to the message area. As in the other sections, you will end by testing the dialog functionality.

The Steps in This Section

This section takes about 20 minutes to complete. It contains the following steps:

Step #15: Adding the Widgets

Step #16: Creating Property Accessor Methods for the Message Box

Step #17: Adding Behavior to the Popup Push Button

Step #18: Testing the Message Box and Text Box

Where You Are in the Tutorial

Section I: Getting Started and Drawing the Interface

Section II: Working with Menus

Section III: Adding Line-Drawing Functionality

⇒Section IV: Working with Message Box Dialogs

Section V: Generating the Application Code

Step #15: Adding the Widgets

In this step you will add the widgets associated with the dialog: a Push Button, Text Field, and the Message Box dialog itself. At run time clicking on the Push Button will pop up the Message Box. Any text you have typed in the Text Field will appear as its message.

1. Add a Push Button to the lower-left area of the Drawing Editor, as shown in Figure 3-34.

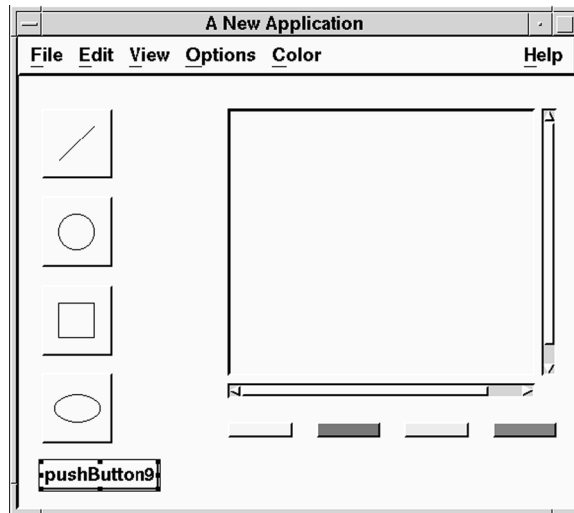


Figure 3-34 Drawing Editor with Push Button Added

2. Load the Push Button into the Property Editor, and change its Label-String property (in the Specific category) from "pushButton9" to "Popup...".
3. Apply your changes.

- Next, add a Text Field to the interface, placing it beside the Push Button, as shown in Figure 3-35.

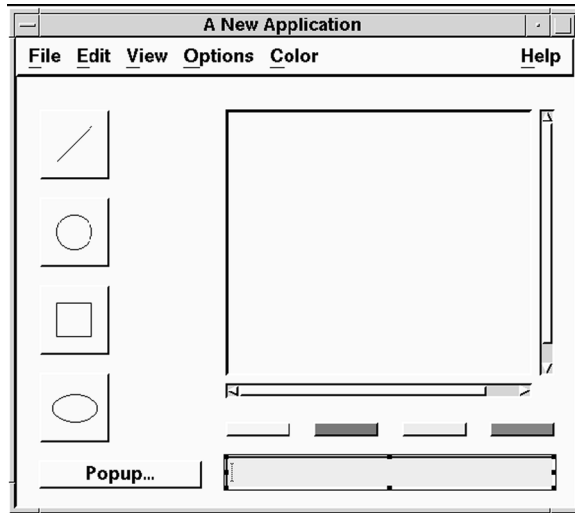


Figure 3-35 Drawing Editor with Text Field Added

- Finally, add a Message Box dialog by clicking on it in the Dialogs area of the Palette and dragging and drawing *outside* the Drawing Editor interface.
- Load the Message Box into the Property Editor, and change its Dialog-Title property (in the Specific category) to "Popup Message".
- Apply your changes.
- Save your work.

Step #16: Creating Property Accessor Methods for the Message Box

In this step you will use the Method Editor to create a pair of *get* and *set* accessor methods for the Message Box. These methods will operate on the Message Box's `MessageString` property. You will also add an instance of the Message Box to the Drawing Editor interface, making the methods easily available to callbacks in the Drawing Editor.

To create property accessor methods for the message box:

1. Select the Message Box dialog, then open the Method Editor for the interface by choosing `Selected Objects`⇒`Tools`⇒`Method Editor`.
2. The Method Editor appears, as shown in Figure 3-36.

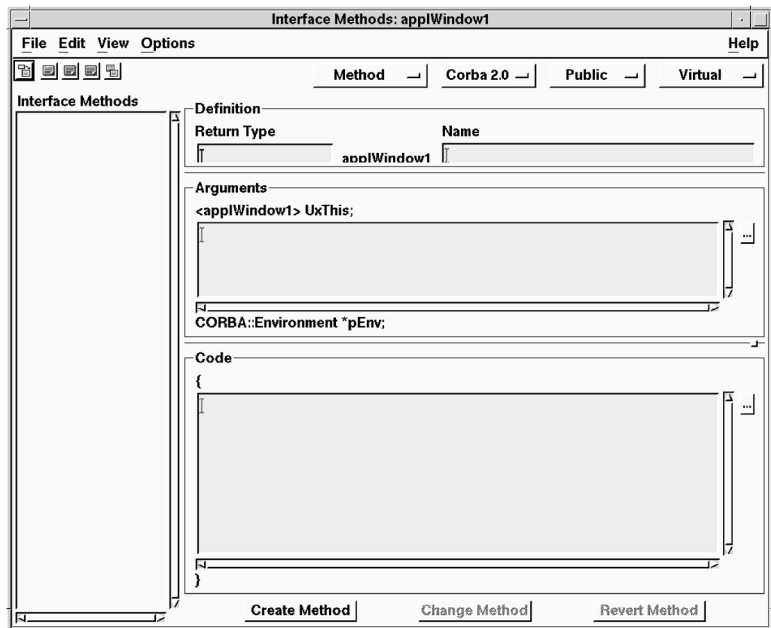


Figure 3-36 Method Editor for the Message Box Dialog, `messageBoxDialog1`

3. Change the method type from `Method` to `GetProperty`. Notice the method prototype changes to reflect the required naming convention. For example, the method name prefix changes from `messageBoxDialog1` to `messageBoxDialog1_get`.

4. Edit a *get* method for the interface by entering the values shown in Table 3-15.

Table 3-15 Get Method Definition

In This Area	Type the Following Code
Return Type	char *
Name	MsgStrng
Arguments	<i>none.</i>
Code	return UxGetMessageString(UxThis);

5. Create the new method by clicking on Create Method.
The *get* method appears in the Interface Methods area, as shown in Figure 3-37.

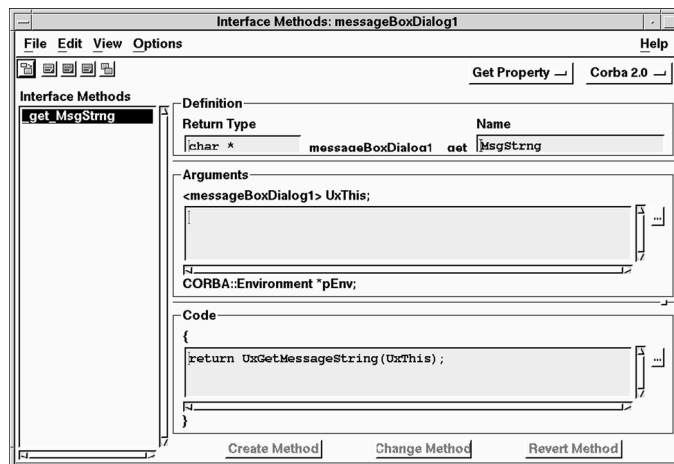


Figure 3-37 Method Editor Showing New Method

6. Similarly, create a *set* method by changing to the Set Property method type.
Notice the method prototype changes once again. For convenience, the Method Editor retains much of the code you entered for the *get* method. A variable called *value* is automatically declared for the *set* method.
7. Edit a *set* method for the interface by entering the values shown in Table 3-16.

Table 3-16 Set Method Definition

In This Area	Type the Following Code
Return Type	void
Name	MsgStrng
Arguments	char *value; (Be sure to change <i>int</i> to <i>char *</i> .)
Code	UxPutMessageString(UxThis, value);

8. Create the new method by clicking on Create Method. The *set* method is added to the Interface Methods area.
9. Close the Method Editor by selecting File⇒Close.
10. To add an instance of the Message Box to the Drawing Editor interface, begin by selecting the Message Box and choosing Selected Objects⇒Instance.
11. Point to the Drawing Editor interface then click the Adjust mouse button. This adds a default-sized instance of the Message Box to the interface. While instances of dialogs are visible in the Browser, they are not visible in the interface itself.
12. Save your work.

Step #17: Adding Behavior to the *Popup* Push Button

In this step you will use the Connection Editor to add behavior to the *Popup* Push Button. First you will use the Connection Editor to attach the Push Button's `ActivateCallback` event—the event that takes place when it is clicked—to the method you created for the Message Box. You will use the method to copy text from the Text Field to the Message Box's Message String. Similarly, you will connect the `ActivateCallback` event to the `UxManage()` method.

1. Select the *Popup... Push Button* just added to the interface.
2. Open the Connections Editor by selecting Selected Objects⇒Tools⇒Connection Editor.

The Connection Editor appears loaded with `pushButton9`, as shown in Figure 3-38.

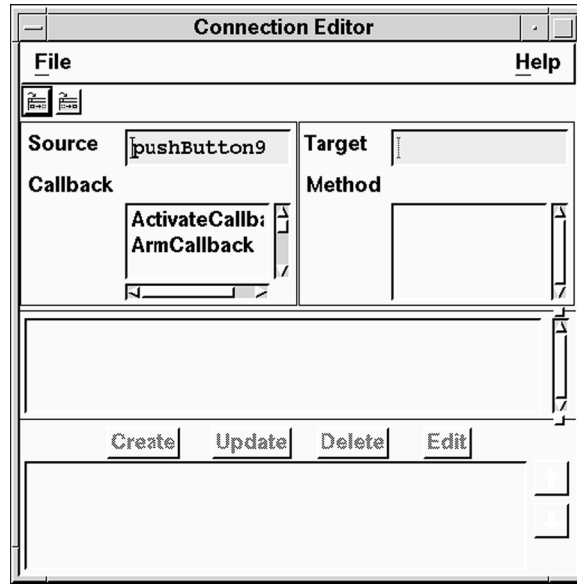


Figure 3-38 Connection Editor

3. Open the Browser by selecting Selected Objects⇒Tools⇒Browser. Since the instance of the Message Box is not visible, you must select it using the Browser.
4. In the Browser, locate the instance of the Message Box, `messageBoxDialog1Instance1`, and load it into the Target area of the Connection Editor by dragging and dropping.

The `_set_MsgStrng` and `_get_MsgStrng` methods you created are displayed in the Method area of the Connection Editor, along with the instance's default methods, as shown in Figure 3-39.

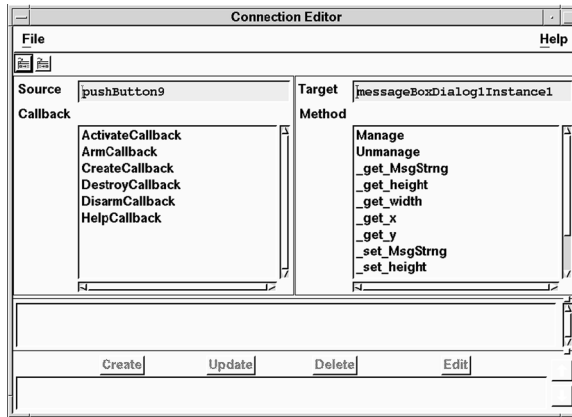


Figure 3-39 Connection Editor Showing Instance's Method

- Click on `ActivateCallback` in list of callbacks, and on `_set_MsgStrng` in the list of methods.

The list of arguments appears in the arguments area.

- Type the following in the value property:
`UxGetText(textField1)`
- Complete the connection by clicking on `Create`.
- The new connection appears, as shown in Figure 3-40.

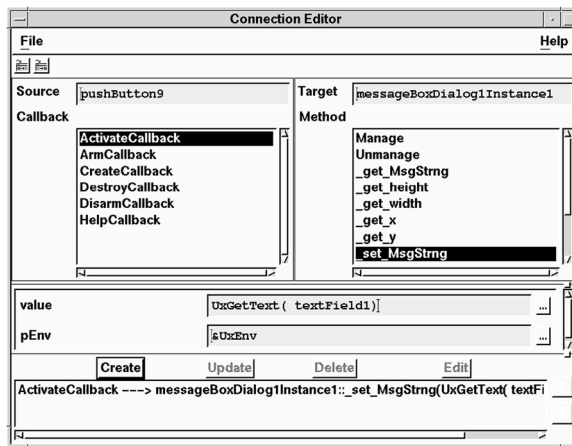


Figure 3-40 Connection Editor Showing New Connection

9. Next, create the connection to pop up the Message Box Dialog by selecting `ActivateCallback` in the Source Callback list, and on `Manage` in the Target Method list.
10. Complete the connection by clicking on `Create`. The new connection is added to the list.
11. Close the Connection Editor and the Browser.
12. Save your work.

Step #18: Testing the Message Box and Text Box

Before generating code for the entire project in the next step, switch to Test Mode to verify the behavior of the portions just added.

1. Switch to Test Mode by clicking on the Test icon in the Project Window.
2. Test the Message Box functions:
 - Pop up the Message Box by clicking on the `Popup...` button.
 - Click on `OK` to pop down the Message Box.
 - Any message you type in the Text Field appears in the Message Box when it pops up.
3. When you are through, switch back to Design Mode by clicking on the Design icon .

Section V: Generating the Application Code

The final step in the Drawing Editor is to generate the application code for the project. Before you can generate the code, you must edit main program to initialize the Drawing Editor draw functions included with the start-up project. You must also edit the Makefile template, to include the object file for the graphics code.

The Steps in This Section

This section take about 15 minutes to complete. It contains the following steps:

Step #19: Customizing the Main Program and Makefile

Step #20: Generating the Code and Running the Executable

Where You Are in the Tutorial

Section I: Getting Started and Drawing the Interface

Section II: Working with Menus

Section III: Adding Line-Drawing Functionality

Section IV: Working with Message Box Dialogs

⇒Section V: Generating the Application Code

Step #19: Customizing the Main Program and Makefile

In this step you will modify the main program to initialize the graphics code in `graphics.c`, the file containing the code for the line-drawing Push Button. You will also edit the makefile template for the project, so that running `make` generates the object file for the code.

Editing the Main Program

In this step you add initialization code to the main program file.

1. Open the Program Layout Editor by choosing `Tools⇒Program Layout` from the Project Window.

The Program Layout Editor appears, as shown in Figure 3-41.

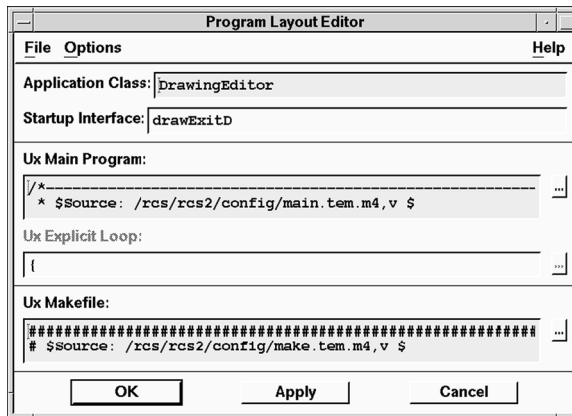


Figure 3-41 Program Layout Editor

2. Open a Text Editor on the main program by clicking on the button next to the Ux Main Program area.

The Text Editor appears as shown in Figure 3-42.

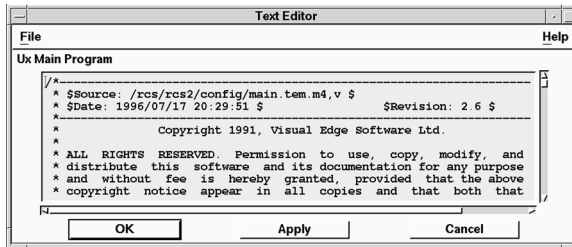


Figure 3-42 Main Program Text Editor

3. Locate the section for global declarations:

```
/*-----
 * Insert application global declarations
 here-----*/
```

4. Just after it, add a declaration for UxInitGraphics:

```
extern int UxInitGraphics();
```

5. Next, locate the section for initialization code:

```
/*-----
 * Insert initialization code for your application
 here-----*/
```

6. Add the following call to UxInitGraphics:

```
UxInitGraphics();
```

7. Click on OK in the Text Editor to complete the change. The Text Editor disappears from view.

Editing the Makefile Template

When generating code, UIM/X uses a makefile template, replacing variables in the template with the names of elements in your project. Since UIM/X cannot know about the code file `graphics.c` used by your project, you must edit the makefile and add names of object files you want produced.

In this step you will edit the makefile template for your project, adding the name of the object file, `graphics.o`.

1. Click on the Text Editor button [...] next to the Ux Makefile field. The Text Editor appears, as shown in Figure 3-43.

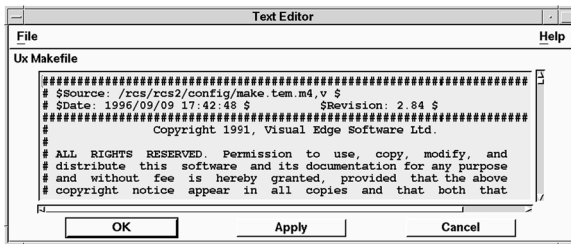


Figure 3-43 Makefile Text Editor

2. Locate the line that begins `APPL_OBJS` and, placing the cursor at the end, add the following:


```
APPL_OBJS = ... graphics.o
```

 For clarity, the part you type is indicated in bold. The three dots indicate you should leave the rest of the text as is. Do not type the three dots.
3. Close the Text Editor by clicking OK.
4. Save your changes and close the Program Layout Editor by clicking OK.
5. Save your work.

Step #20: Generating the Code and Running the Executable

The final step in creating your project is to generate code for the Drawing Editor.

1. Check that you are in Design Mode.

2. Choose Options⇒Code Generation in the Project Window. The Code Generation Options window appears, as shown in Figure 3-44.

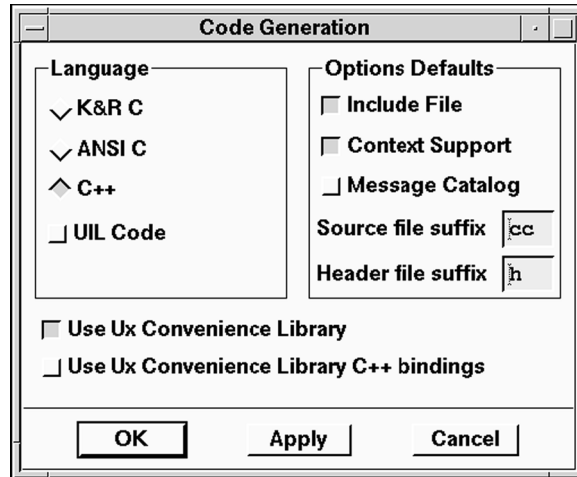


Figure 3-44 Code Generation Options

3. Check that the language selected is ANSI C.
If you wish to generate C++ code, copy or rename `graphics.c` to `graphics.cc` in a terminal window.
4. Save your changes and close the dialog by clicking on OK.
5. Click on the Run icon in the Project Window's icon bar.
UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
6. Test your program. Verify that it works as it did in Test Mode.
7. To stop the program choose File⇒Exit.
8. Switch back to Design mode by clicking on the Design icon
9. Save the changes to your program.
Now when you modify the Drawing Editor, you can simply click on the Run Mode toggle to generate the code, compile it, and run the executable in one step.

Where to Go From Here...

Congratulations on having completed the Drawing Editor tutorial! If you wish to continue to develop the Drawing Editor, additional functionality can easily be added. The start-up project contains action routines not yet used in your project. Table 3-17 lists the functionality you can add, along with the icons and action routines required.

Table 3-17 Additional Functionality You Can Add

Functionality	Icon	Action Routine
Drawing Freehand	freehand.xpm	freehand
Writing Text	draw_text.xpm	draw_text

For other ideas, two project files have been included. The `draw_final.prj` project file contains the completed Drawing Editor. The `draw_full.prj` project file contains a Drawing Editor tutorial with the additional freehand and text functionality.

Building a GUI for a Command-Line Application

4

Overview

Building a GUI for a command-line application is a three-step process. First you lay out the interface. Next you add subprocess control code to the interface using the Declaration Editor. Finally, you add callback behavior to execute the subprocess on the appropriate user action.

The UIM/X Convenience Library features a number of functions especially designed for subprocess control. `UxCreateSubproc()`, for example, creates the subprocess, returning a handle to it. You can then run the subprocess by calling `UxExecSubproc()`.

More often than not, command line applications permit (or require) that you specify options at the command line. UIM/X provides a number of ways to present and submit any arguments that might be expected. Options menus are convenient for presenting mutually exclusive options, with only the currently active option visible. Toggle Buttons can be used to display all available options, and can be mutually exclusive or permit multiple selection. The simplest way to build the command is to append the selected options to a string defined globally for the interface.

The GUI You Will Build

This chapter demonstrates how to use UIM/X in Standard Mode to create an interface for a command-line application, illustrating subprocess control. The Command Line interface, shown in Figure 4-1, executes the UNIX `ls` command as a subprocess to list the contents of directories. You select arguments for the command graphically, using Toggle Buttons and an Option menu.

The interface consists of the following elements:

- *Text Field*: A Text Field where the user can enter the directory to be listed.
- *Toggle Buttons*: Mutually exclusive Toggle Buttons for selecting the file attributes listed.
- *Option Menu*: An Option menu for listing files alphabetically, reverse alphabetically, by latest date, or earliest date.
- *Scrolled Text*: A scrollable window showing the results of the `ls` subprocess.
- *OK Push Button*: The Push Button that spawns the subprocess.

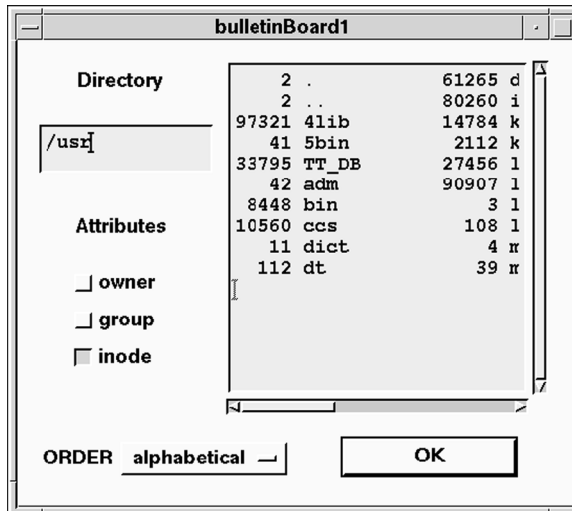


Figure 4-1 The Completed Command Line Project

The Steps in This Tutorial

This tutorial takes about 45 minutes to complete. It contains the following steps:

- Step #1: Starting UIM/X in Standard Mode
- Step #2: Laying Out the Interface
- Step #3: Changing Labels and Other Properties
- Step #4: Adding Declarations and Final Code
- Step #5: Adding Behavior to the Interface
- Step #6: Testing the Program
- Step #7: Generating the Code and Running the Executable

Step #1: Starting UIM/X in Standard Mode

Before you begin this tutorial, set up a new directory as follows:

1. Start the X Window System.
2. Bring up a terminal window.
3. Make a directory to store the files you will create in this tutorial:

```
mkdir chap4
```

4. Change to the directory you just created:

```
cd chap4
```

5. Start UIM/X from your new directory:

```
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 &
```

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and the Palette appear.

6. Iconify the terminal window.

Note: To restart the tutorial, begin again from Step 4 above.

Step #2: Laying Out the Interface

In this step you will lay out the interface for the Command Line project. First, you will create a Bulletin Board to contain the other widgets. Then, you will add a Text Field where the files will be displayed, a Text widget for entering the directory to be listed, and Labels to identify the areas, and a Push Button. Next you will add Toggle Buttons for selecting the information displayed in the Text Field. Finally, you will add an Option menu for ordering the files.

Drawing the Bulletin Board

In this step you will drag and draw a Bulletin Board widget.

1. Make sure you are in Design Mode. If not, click on the Design toggle button.
2. In the Managers area of the Palette, click on the Bulletin Board icon.
3. Move the mouse pointer to where you want the upper-left corner of the Bulletin Board to appear.
4. Press and hold the Select mouse button, then drag the mouse downwards and to the right to draw the Bulletin Board. To complete the operation, release the mouse button.

The Bulletin Board widget appears as shown in Figure 4-2.

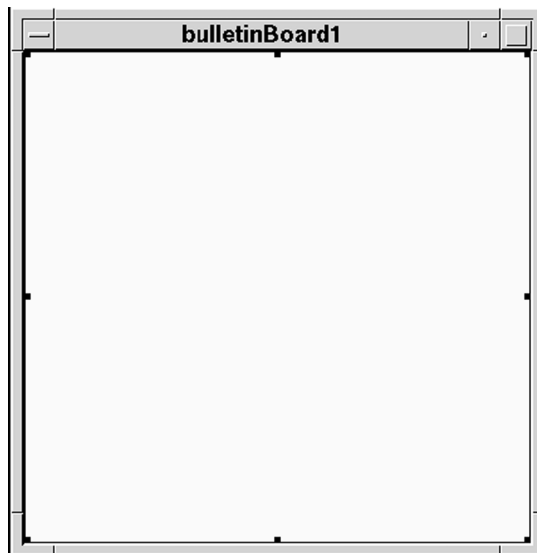


Figure 4-2 The Bulletin Board Widget

Adding the Text, Push Button, and Label Widgets

In this step you will add the remaining widgets that make up the Command Line project's interface. You will add a Text, a Scrolled Text, Push Button, and Label. You will also duplicate the Label. At the end of this step the interface should look similar to Figure 4-3.

1. In the Primitives category of the Palette, click on the Text icon with the Adjust mouse button.
Be sure to select the Text icon, and not the Text Field icon.
2. Drag the outline of the widget onto the Bulletin Board, and release it in the upper left corner.
3. Similarly, in the Primitives category click on the Scrolled Text icon, then drag and drop the widget, placing it on the right side of the Bulletin Board.
4. Resize the Scrolled Text until it fills most of the right hand side of the interface, using Figure 4-3 as a model.
5. Add a Push Button by dragging and dropping, placing it below the Scrolled Text.
6. Add a Label to the interface, placing it above the Text widget.
7. To duplicate the Label, begin by pressing the Menu mouse button to display the Selected Objects popup menu.
8. Choose Duplicate to make a copy of the first Label, then drag and drop it below the Text widget.
9. Save your work as a new project, `CommandLine.prj`.

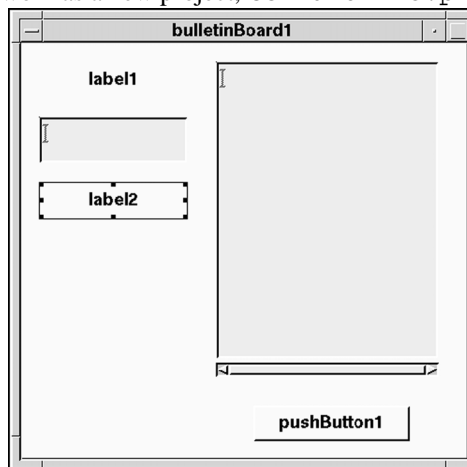


Figure 4-3 The Text, Push Button, Scrolled Text, and Label Widgets Added

Creating the Row Column and Toggle Buttons

The Command Line project features Toggle Button gadgets used to change the file attributes displayed in the Scrolled Text. In this step you will add the Toggle Buttons, placing them inside a Row Column. Row Columns make excellent containers for several widgets of the same type, since they can position the widgets in a grid.

1. In the Managers category, click on the Row Column icon.
2. Place the Row Column below the second Label, `label2`, as shown in Figure 4-4.

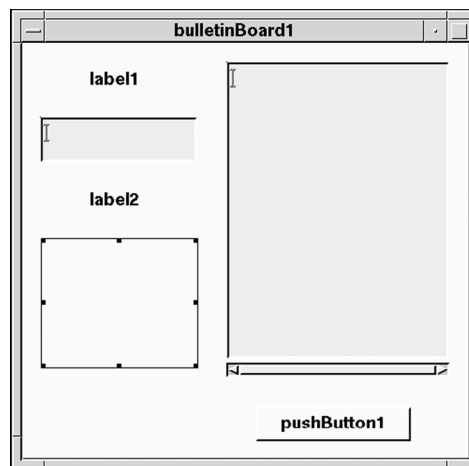


Figure 4-4 Bulletin Board with Row Column Added

3. In the Gadgets category of the Ux Palette, click on Toggle Button Gadget.
4. Position the toggle button on the top part of the Row Column widget.
Notice how the Row Column automatically shrinks to fit the Toggle Button. This is the expected behavior.
5. Duplicate the Toggle Button by choosing Selected Objects⇒Duplicate. To display the Selected Objects popup menu, press the Menu mouse button.
6. In the same way, create a third Toggle Button by duplication.

Note the new Toggle Buttons are automatically placed below the first one, as shown in Figure 4-5.

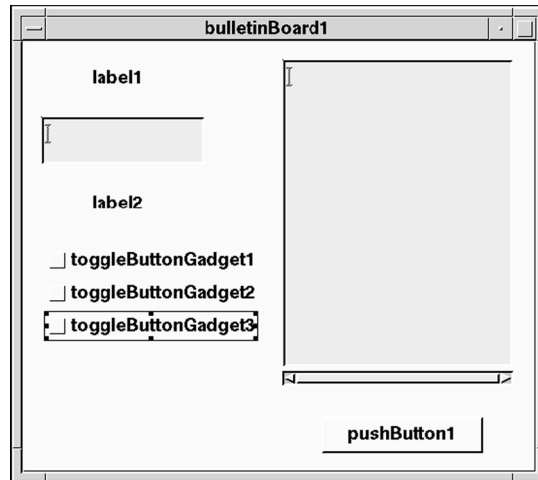


Figure 4-5 Bulletin Board with All Three Toggle Buttons Added

7. Save your work.

Creating the Option Menu

Like Toggle Buttons, Option Menus are a convenient way to present a limited number of choices to the user. Unlike toggle buttons, only the current option is displayed by the Option Menu.

In this step you will add an Option Menu to the interface. The menu will be used to specify the order in which files are displayed: alphabetical, reverse alphabetical, latest first, or earliest first.

1. In the Menus category, click on the Option Menu icon.
2. Position the menu below the third toggle button, but outside the Row Column.

Notice it appears with an option button already in place.

3. Double-click on the Option Menu to open the Option Menu Editor.
4. Select `optionMenu_p1` in the Panes list, then type "ORDER" in its `LabelString` property.
5. Display the properties for the first item in the menu, `optionMenu_p_b1`, by clicking on it in the Items list.
6. Change its `Label String` property to "alphabetical".

7. To add an item to the menu ensure `optionMenu_p_b1` is selected in the Items list, then choose Create⇒Item After⇒Push Button.
8. Change its Label String property to "reverse alpha".
9. Repeat the process to add another Push Button item to the menu, this time with Label String property set to "latest first".
10. Create one last Push Button item, with LabelString property set to "earliest first".

When complete, the Option Menu Editor should appear as shown in Figure 4-6.

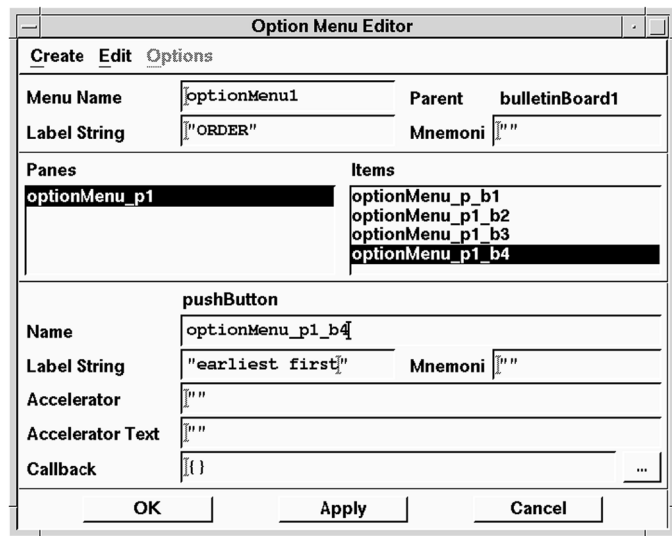


Figure 4-6 Option Menu Editor Showing All Four Items

11. Click on OK in the Option Menu Editor to apply the changes to the interface.

The interface is updated to reflect the changes, as shown in Figure 4-7.



Figure 4-7 All the Widgets in Position

12. Save your work.

Step #3: Changing Labels and Other Properties

Now that the widgets are in place, you are ready to change their titles, labels and other properties. In this step you will begin by changing the Scrolled Text to display multiple lines of information (by default it scrolls horizontally). Then you will change the default string displayed in the Message Box. In UIM/X you change properties at design time using the Property Editor.

1. Double-click on the Scrolled Text to open the Property Editor and load the widget into it in one step.
2. Locate the `EditMode` property in the `Specific` category.
3. Click on the `EditMode` option menu, changing it to `multi_line_edit`, as shown in Figure 4-8.

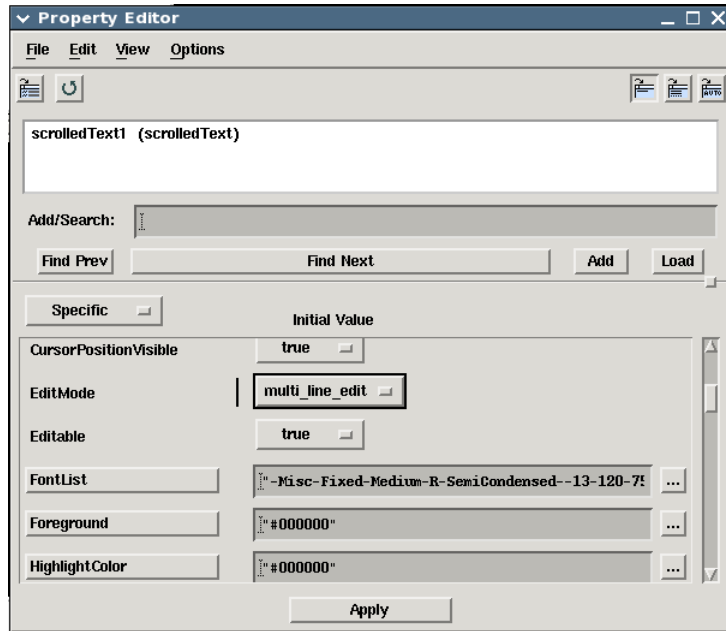


Figure 4-8 Property Editor Loaded with the Scrolled Text Widget

4. Apply the change by clicking on Apply.
5. Set the Property Editor to load widgets automatically by choosing Options ⇒ Automatic Load.

Now you can load a widget into the Property Editor simply by selecting it.

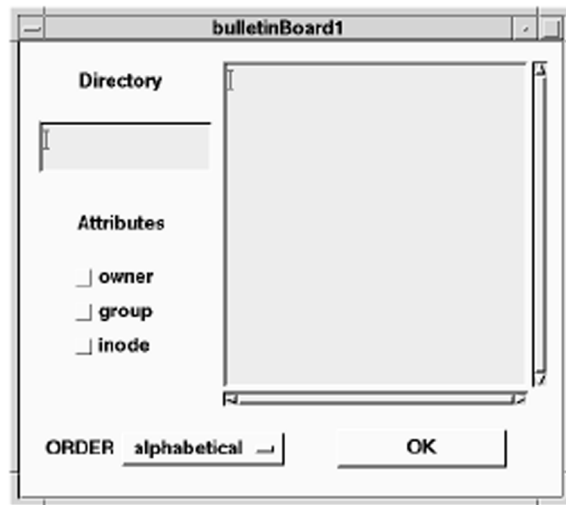
6. Click on `pushButton1` to load it into the Property Editor.
7. Locate the `LabelString` property in the `Specific` category, and replace "`pushButton1`" with "OK".
8. Apply the change by clicking on Apply in the Property Editor.
9. Continue loading widgets into the Property Editor (using automatic loading), and changing their `LabelString` properties.

Table 4-1 lists all the widgets whose `LabelStrings` must be changed, and the values you should give them.

Table 4-1 Widgets and New LabelString Properties

Widget Name	New LabelString Property
label1	"Directory"
label2	"Attributes"
toggleButtonGadget1	"owner"
toggleButtonGadget2	"group"
toggleButtonGadget3	"inode"

When you have made the changes, your interface should appear as shown in Figure 4-9.

*Figure 4-9* Bulletin Board with LabelStrings Changed

10. Save your work.

Step #4: Adding Declarations and Final Code

In this step you will open the Declaration Editor for the Bulletin Board and define a few global constants. These will be shared by the interface's callbacks that you will add later.

1. Click on the Bulletin Board to select it.
2. Open the Declaration Editor by pressing the Menu mouse button while over the interface, and choosing Selected Objects⇒Tools⇒Declaration Editor.
3. The Declaration Editor appears, as shown in Figure 4-10.

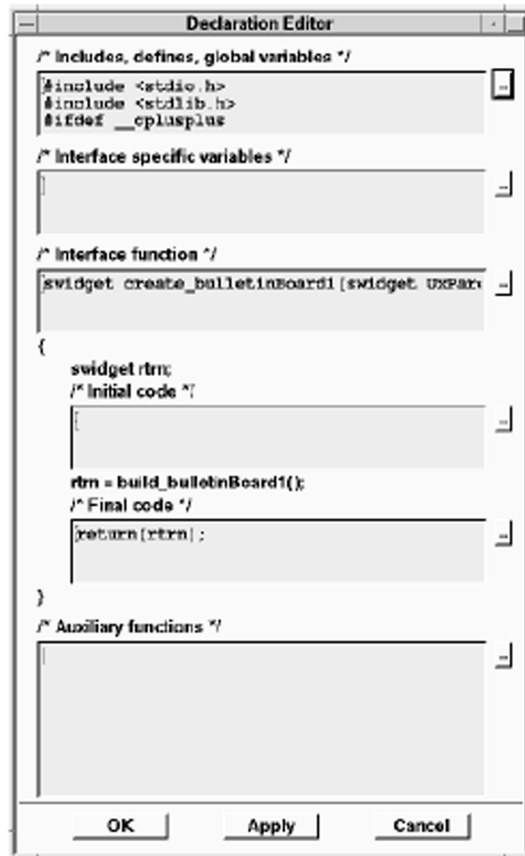


Figure 4-10 Declaration Editor

4. Open the Text Editor by clicking on the [...] button next to the /* Includes, defines, global variables */ area, and add the following constants:

```
#include "UxSubproc.h"
#include "UxLib.h"

int Owner = False,
    Group = False,
    Inode = False;

char Attribute[10] = "-a ";
handle h;
```

Make sure to enter a space for the character definition ("-a "). During the subprocess this is passed to UNIX, which requires the space to run the command line properly.

5. Click on OK to close the Text Editor.
6. Open the Text Editor for the `/* Final Code */` area and add the following code, just before the return call (the return call is shown below, for reference).

```
/* create subprocess object */
h=UxCreateSubproc("/bin/ls ", "", UxAppendTo);
if (UxSetSubprocClosure( h, (char *) UxGetWidget
    (scrolledText1 ) ) == -1)
{
    printf("Can't set subprocess closure\n");
}
return(rtrn);
```

As above, make sure to include the space indicated ("/bin/ls ").

7. Click on OK in the Text Editor.
8. Click on OK in the Declaration Editor.
9. Save your work.

Step #5: Adding Behavior to the Interface

Now that you have defined global variables and added final code to the interface you are ready to add behavior to the widgets by specifying callbacks. First you will add behavior to the Toggle Buttons using the Property Editor. Next you will add behavior to the Option menu using the Option Menu Editor. Finally, you will add behavior to the OK Push Button.

Adding Behavior to the Toggle Buttons

In this step you will add callback code to the Toggle Buttons. Since all three Toggle Buttons contain similar callback code, you will load them all into the Property Editor at the same time, write the callback, then customize it for each Toggle Button.

1. Click on the *owner* Toggle Button to load it into the Property Editor.
2. Load the group and inode Toggle Buttons into the Property Editor with the first, by holding down the Control key and clicking each one in turn.
3. Locate the `ValueChangedCallback` property in the Behavior category, and open the Callback Editor by clicking on the Text Editor button (...) beside it.
4. Click in the Text Field, and type the following code exactly as it appears:

```
XmToggleButtonCallbackStruct *s
    = (XmToggleButtonCallbackStruct *)
    UxCallbackArg;

Owner = s->set;
```

5. Click on OK in the Callback Editor.
6. Click on Apply in the Property Editor.
7. Click on the *group* Toggle Button to load it alone into the Property Editor.
8. Open the Callback Editor for `ValueChangedCallback`, changing `Ownerto Group`.
9. Click on OK in the Callback Editor, then click on Apply in the Property Editor.
10. Repeat the process for the *inode* Toggle Button, replacing `Owner` with `Inode`.
11. Save your work.

Adding Behavior to the Option Menu Buttons

In this step you will use the Option Menu Editor to add behavior to the Option Menu buttons.

1. Double-click the Option menu button (labelled alphabetical) to pop up the Option Menu Editor.
2. Click on the first Push Button in the Items list, `optionMenu_p_b1`, and enter the following callback for it:

```
strcpy(Attribute, " -a ");
```


- Click on each of the remaining Push Buttons on the Items list and enter the Callbacks listed in Table 4-2:

Table 4-2 Callback Code for Option Menu Items

Item	Callback Code
optionMenu_p1_b2	<code>strcpy(Attribute, " -ar ");</code>
optionMenu_p1_b3	<code>strcpy(Attribute, " -t ");</code>
optionMenu_p1_b4	<code>strcpy(Attribute, " -tr ");</code>

Make sure to enter a space before *and* after each parameter to be passed to UNIX, i.e. " -ar ", " -t ", and " -tr ". After making your entries, the Option Menu Editor should look as shown in Figure 4-11.

- Apply your changes and close the Option Menu Editor by clicking on OK.

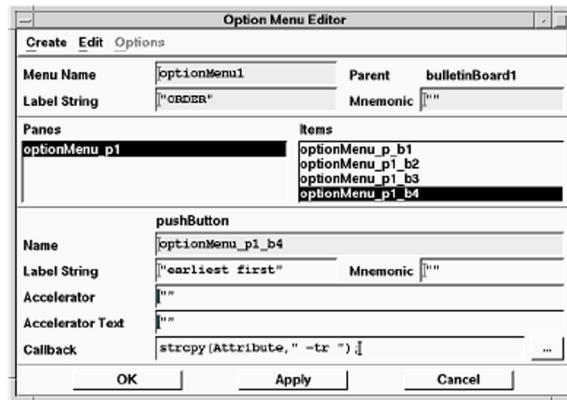


Figure 4-11 Option Menu Editor with the New Callbacks

- Save your work.

Adding Behavior to the OK Push Button

In this step you will add behavior to the OK Push Button.

1. Load the OK Push Button into the Property Editor.
2. Open the Callback Editor by clicking on the Text Editor button (...) beside ActivateCallback (in the Behavior category).
3. Click in the Text Field, and type the following code:

```
char arglist[128];
UxClearText(scrolledText1);
arglist[0] = '\0';
/* owner, group & inode flags */
if (Owner)
    strcat(arglist, " -o ");
if (Group)
    strcat(arglist, " -g ");
if (Inode)
    strcat(arglist, " -i ");
/* alphabetically, ... */
strcat(arglist, Attribute);
/* directory name */
strcat(arglist, UxGetText(text1));
if (UxExecSubproc(h, arglist) == -1)
{
    printf("Can't start the application\n");
    return;
}
```

As before, don't forget to a space before and after each parameter passed to UNIX, as indicated.

4. Click on OK in the Callback Editor.
5. Click on Apply in the Property Editor.
6. Save your work.

Step #6: Testing the Program

Before generating code for the project in the next section, take a moment to switch to Test Mode.

1. Switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

2. Hide the Command Line interface by choosing View⇒Hide Project.

Note: This will cause the “CommandLine” application’s GUI to disappear in Test Mode. This is the desired response, as the subsequent steps will re-launch the application through the Interpreter. The more conventional Test Mode interface lacks the needed subproc support, but loading the module into the Interpreter will take care of this detail while in Test Mode.

3. Choose Tools⇒Interpreter in the Project Window.
4. Select the Bulletin Board interface by clicking on its icon in the Project Window.
5. Choose Module⇒Selected Interface in the Interpreter or click on the corresponding icon .

The Interpreter title bar changes to reflect the new scope.

6. Enter the following code in the Interpreter window.

```
UxPopupInterface (create_bulletinBoard1 (NO_PARENT)  
, no_grab);
```

7. Triple-click the line of code to highlight it, then choose Interpret⇒Evaluate.

The Interpreter evaluates the code, pops up the Command Line interface, and prints the following to the Interpreter Messages Area:

```
Result: 0
```

8. Close the Interpreter by choosing File⇒Close.

9. Test the interface:
 - Type a directory name in the Directory Text widget.
 - Click on a Toggle Button to view files listed by owner, group, or inode.
 - Order the files alphabetically, reverse alphabetically, earliest first, or latest first.
 - Click on OK to display the files.
10. When you are through, switch back to Design Mode by clicking the Design icon
When prompted to do so, discard the duplicate interfaces.

Step #7: Generating the Code and Running the Executable

You have now successfully constructed and tested the Command Line project. In this step you will generate the code for the application, and run it, without leaving the development environment.

1. Check that you are in Design Mode. If not, click on the Design icon
2. Choose Options⇒Code Generation on the Project Window menu.
3. In the Code Generation options window that appears, check that the language selected is ANSI C, and that Context Support is deselected.
More economical code is produced with context support disabled, for applications using only single copies of an interface.
4. Save your changes and close the dialog by clicking on OK.
5. Click on the Run icon in the Project Window's icon bar.
6. Click OK to generate your code.
UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
7. Test your program. Verify that it works as it did in Test Mode.
8. To stop the program select Close from the window control box.
9. Save the changes to your program.

Now when you modify the Command Line project, you can simply click on the Run Mode icon to generate the code, compile it, and run the executable in one step.

Part III: Advanced Tutorials

Overview

The tutorials of this section are of an advanced nature. These tutorials are
Creating an RGB Color Editor in C++ and Integrating a Non-Visual Object.

Creating an RGB Color Editor in C++

5

Overview

In a sophisticated application it is possible to spend a great deal of time perfecting a small portion of the interface. In such cases it might be desirable to reuse that portion in other areas, at the same time protecting it from unwanted changes. As an interface management system that encourages object-oriented development, UIM/X provides a simple yet robust mechanism for doing so. You create a new class for the portion to be reused, then add an instance of the class to the interface.

In UIM/X each stand-alone interface is its own class. To create a new class, you simply turn the widget into a stand-alone interface by dragging and dropping it onto the desktop. To reuse it in another interface, UIM/X lets you drag and draw (or drag and drop) an *instance* of the class on the target interface.

Since they inherit all their properties and behavior from their class, instances are exact duplicates of the originating class. Properties and behavior are by default uneditable in the instance, providing the desired protection from unwanted changes. Should you want to, it is a simple matter to expose properties or behavior in the instances.

To expose properties in an instance you define property accessor methods for the class. These are paired *get* and *set* methods following a specific naming convention. In the body of the methods, you write code that acts on the property you want to expose. UIM/X recognizes pairs of property accessor methods, and presents the new property in the Specific category of the Property Editor.

Similarly, you can create behavior accessor methods to present callbacks in the Property Editor. A behavior accessor method is a single method following a specific naming convention. In this case, the body of the method contains code that adds the desired Xt callback to the callback list. UIM/X presents the new callback in the Behavior category of the Property Editor.

Exposed properties and new callbacks behave just like those provided by default. As noted, they show up in the Property Editor in the appropriate category. You can provide a new property with a default value at design time, and write callback code for new callbacks. In addition, you can set properties at runtime in callback code, or connect callbacks to properties graphically using the Connection Editor.

The GUI You Will Build

Note: If you have installed UIM/X in its C-only configuration, do not attempt the tutorial in this chapter. The tutorial assumes that UIM/X is running in C++ mode, and that you have a C++ compiler.

In this chapter you will create an interface to function as an RGB Color Editor. The Color Editor illustrates how to create a new class, expose properties and behavior in instances of the class, and how to call interface methods to update a display (in this case, a sample color). In addition, it provides an example of programming in C++. The completed interface, shown in Figure 5-1, consists of the following areas:

- *Color-Changing Scales:* Horizontal Scales with a range from 0 to 255. One each for the red, green, and blue component of an RGB color definition. Each scale is an instance of a Scale class with properties and behavior exposed.
- *Color Display Area:* A Drawing Area widget whose `BackgroundColor` property is updated by the Scales, using a shared interface method.
- *Information Display Area:* Two Labels, one of which is updated by the above interface method.

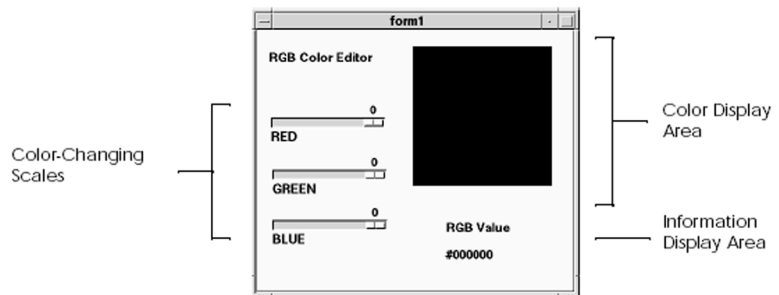


Figure 5-1 The Completed RGB Color Editor Project

The Steps in This Tutorial

This tutorial takes about 60 minute to complete. It contains the following steps:

- Step #1: Starting UIM/X in Standard Mode
- Step #2: Laying Out the Interface
- Step #3: Changing LabelStrings and Other Properties
- Step #4: Adding Declarations and Global Code
- Step #5: Defining a Method to Update the Display
- Step #6: Creating a Scale Class
- Step #7: Exposing Properties in the Scale Class
- Step #8: Exposing Behavior in the Scale Class
- Step #9: Setting Properties in the Instance
- Step #10: Adding Behavior to the Instance
- Step #11: Completing the Interface
- Step #12: Testing the Program
- Step #13: Generating the C++ Code and Running the Executable

Step #1: Starting UIM/X in Standard Mode

Before you begin building the RGB Color Editor project, set up a new directory as follows:

1. Start the X Window System.
2. Bring up a terminal window.
3. Make a base directory for this tutorial:

```
mkdir chap5
```
4. Change to the directory you just created:

```
cd chap5
```
5. Start UIM/X from your new directory:

```
uimx &
```

Note: Since this tutorial features C++ code in its declarations and callbacks, do not specify any language options at the command line. UIM/X starts in C++ mode by default.

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 &
```

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and UIM/X palette appear.

6. Iconify the terminal window.

Note: To restart the tutorial, begin again from Step 4 above.

Step #2: Laying Out the Interface

In this step you will lay out the interface for the Color Editor. First, you will create a Form to contain the other widgets. Then, you will add a Drawing Area, a Scale, and Labels to identify the areas and display color information.

To lay out the interface:

1. Drag and draw (or drag and drop) a Form from the Managers category of the Palette to your desktop, as shown in Figure 5-2.

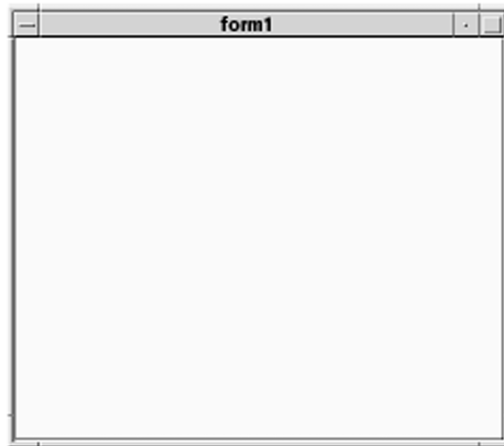


Figure 5-2 The Form Widget

2. Add a Drawing Area (from the Managers category) to the Form, making it large enough to almost fill the upper-right corner of the Form, as shown in Figure 5-3.

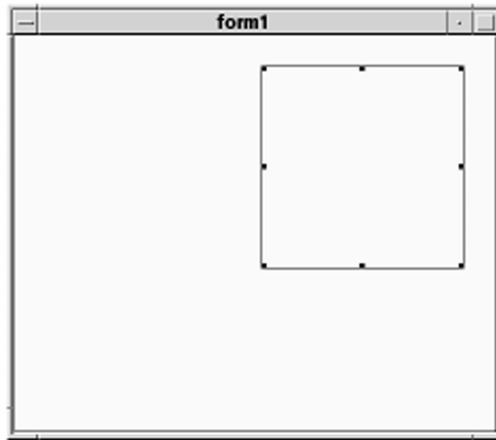


Figure 5-3 Form Widget with Drawing Area Added

3. Add a Label to the interface, placing it in the upper left corner of the Form, as shown in Figure 5-4.

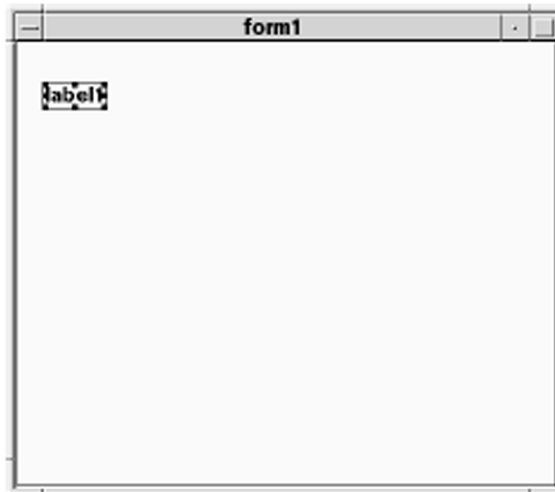


Figure 5-4 Form with Label Added

4. Create two more Labels by dragging and dropping, dragging and drawing, or duplication, positioning them as shown in Figure 5-5.

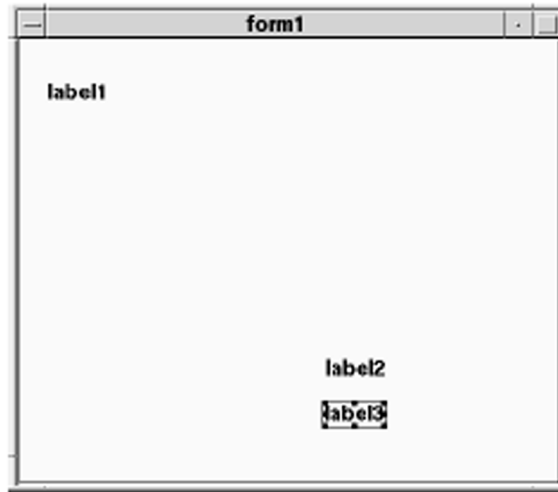


Figure 5-5 Form with All Three Labels Added

5. Finally, add a Horizontal Scale to the interface, positioning it under the first Label.

You will add the other two Horizontal Scales once you have created the Scale class, exposed properties and behavior, and connected it to the interface elements, later.

6. Your interface should look similar to Figure 5-6. (The widgets are shown selected, for display purposes.)

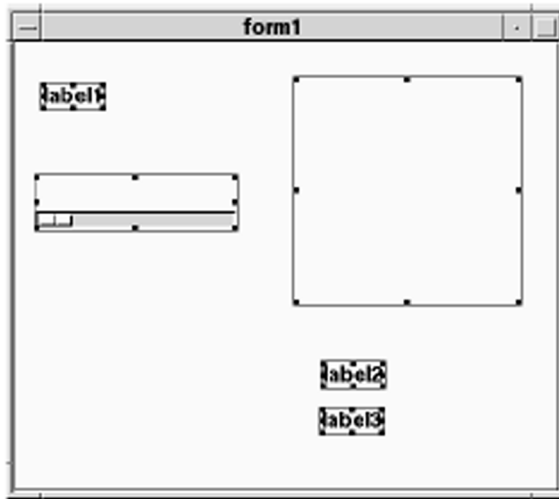


Figure 5-6 Form with All Necessary Widgets

7. Save your work as a project, calling it `ColorEditor.prj`.

Step #3: Changing LabelStrings and Other Properties

Now that the widgets are in place, you are ready to change their `LabelStrings` and other properties.

1. Double-click on the `label1` to open the Property Editor and load the widget into it in one step.
2. Locate the `LabelString` property in the `Specific` category, changing it from `"label1"` to `"RGB Color Editor"`
3. Apply the change by clicking on `Apply`. The interface is updated to reflect the change, as shown in Figure 5-7.

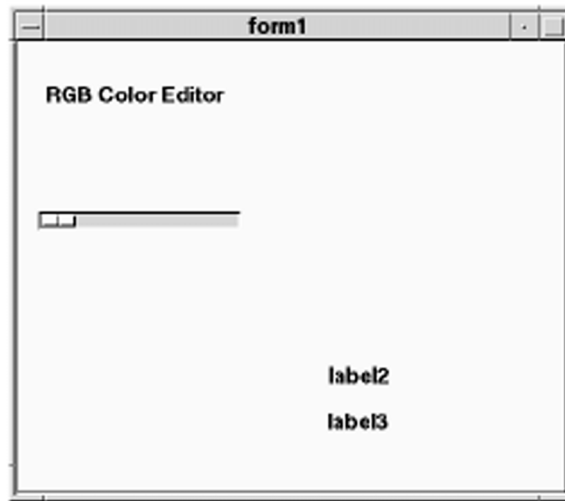


Figure 5-7 RGB Color Editor with New Label

4. Set the Property Editor to load widgets automatically by choosing Options⇒ Automatic Load.
Now you can load a widget into the Property Editor simply by selecting it.
5. Click on `label2` to load it into the Property Editor.
6. Locate the `LabelString` property in the `Specific` category, changing it from `"label2"` to `"RGB Value"`.
7. Apply the change by clicking on `Apply`.
8. Continue loading widgets into the Property Editor (using automatic loading) and changing properties.

Table 5-1 lists all the widgets with properties to be changed, and the values you should give them.

Table 5-1 Property Changes for the Color Editor

Widget Name	Category	Property Name	New Value
<code>label1</code>	<code>Specific</code>	<code>LabelString</code>	<code>"RGB Color Editor"</code>
<code>label2</code>	<code>Specific</code>	<code>LabelString</code>	<code>"RGB Value"</code>
<code>label3</code>	<code>Specific</code>	<code>LabelString</code>	<code>"#000000"</code>
<code>drawingArea1</code>	<code>Core</code>	<code>Background</code>	<code>"black"</code>

Widget Name	Category	Property Name	New Value
scaleH1	Core	Height (set the <i>source</i> —see note below)	Default
scaleH1	Specific	Maximum	255
scaleH1	Specific	Processing Direction	max_on_left
scaleH1	Specific	ShowValue	true
scaleH1	Specific	TitleString	"Color"

Note: In order for the Scale's title to be shown, you must set the *source* of its Height property to Default. (When drawn, the source is automatically set to Private.) To show a widget's property sources, choose View⇒Hide Source in the Property Editor.

When you have made the changes, your interface should appear as shown in Figure 5-8.

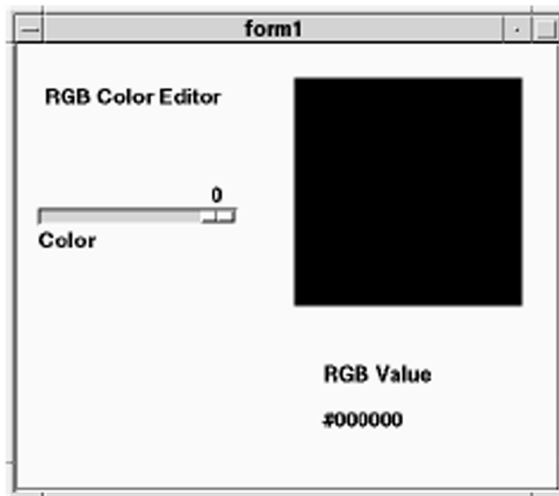


Figure 5-8 Form with LabelStrings and Other Properties Changed

9. Close the Property Editor by choosing File⇒Close.
10. Save your work.

Step #4: Adding Declarations and Global Code

In this step you will enter code to be used by multiple elements in your interface. First you will declare some global variables, and a class called *rgbcolor*. Next you will define interface-specific variables and set initial and final code. Finally you will define the body for the *rgbcolor* class. All work will be performed in the Declaration Editor.

Defining Global Variables and the *rgbcolor* Class

In this step you will declare the *rgbcolor* class, a class containing two public member functions to initialize and update an RGB value respectively. You will also declare a few global variables for use in the interface.

1. Click on the Form to select it.
2. Open the Declaration Editor by pressing the Menu mouse button while over the interface, and choosing Selected Objects⇒Tools⇒Declaration Editor.
3. The Declaration Editor appears, as shown in Figure 5-9.

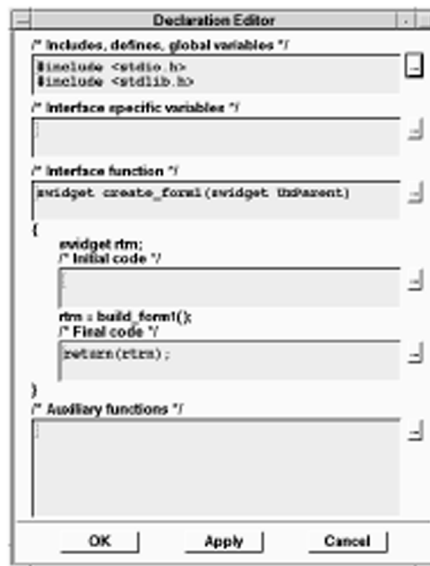


Figure 5-9 The Declaration Editor

4. Click on the Text Editor button (...) beside the
 /* Includes, defines, global variables */.

5. Add the following declarations, just after the `#endif` statement (shown below for reference):

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus
#include <iostream.h>
#endif /* __cplusplus */
#include <X11/Xutil.h>

class rgbcolor
{
private:

    Display *display;
    Colormap cmap;
    int screen;
    XColor newcolor;

public:
    void InitColor(swidget, unsigned short,
        unsigned short,
        unsigned short);
    void UpdateColor(unsigned short, unsigned
        short, unsigned
        short);
};
```

6. Close the Text Editor and copy the code to the Declaration Editor by clicking on OK.
7. Apply your changes to the interface without closing the Declaration Editor by clicking on Apply.
8. Save your work.

Defining Interface-Specific Variables and Initial and Final Code

The Declaration Editor allows you to define interface-specific variables, as well as to set initial code and final code. Initial code is executed when the interface is created. Final code is executed just before the interface is popped up. In this step you will add code to these areas of the Declaration Editor, initializing the Drawing Area's background color to black.

1. Click on the Text Editor button (...) beside the `/* Interface specific variables */` area.

2. Add the following declarations:

```
rgbcolor current_color;  
unsigned short v1, v2, v3;
```

3. Click OK in the Text Editor.

4. Add the following code to the `/* Initial Code */` area.

```
v1 = v2 = v3 = 0;
```

5. Add the following code to the `/* Final Code */` area, just before the return call (the return call is shown below, for reference).

```
current_color.InitColor(drawingArea1, v1, v2,  
v3);  
return(rtrn);
```

6. Click on Apply in the Declaration Editor.

7. Save your work.

Defining Auxiliary Functions

In this step you will define `rgbcolor`'s two public functions:

`InitColor()` and `UpdateColor()`.

1. Click on the Text Editor button (...) beside the `/* Auxiliary functions */` area.

2. Add the following definitions:

```
void rgbcolor::InitColor( swidget sw, unsigned
    short red, unsigned short green, unsigned short
    blue )
{
    Arg arglist[20];
    unsigned long planes[1], pixels[1];
    display = XtDisplay( UxGetWidget(sw) );
    screen = DefaultScreen( display );
    cmap = DefaultColormap( display, screen );
    XAllocColorCells( display, cmap, False,
planes, (int) 0, pixels, (int) 1 );
    newcolor.flags = DoRed | DoGreen | DoBlue;
    newcolor.pixel = pixels[0];
    newcolor.red = red;
    newcolor.green = green;
    newcolor.blue = blue;
    XStoreColor( display, cmap, &newcolor );
    XtSetArg(arglist[0], XmNbackground,
pixels[0]);
    XtSetValues( UxGetWidget(sw), arglist, 1);
}

void rgbcolor::UpdateColor( unsigned short red,
    unsigned short green,
    unsigned short blue )
{
    newcolor.red = red * 65535 / 256;
    newcolor.green = green * 65535 / 256;
    newcolor.blue = blue * 65535 / 256;
    XStoreColor( display, cmap, &newcolor );
}
```

3. Click on OK in the Text Editor.
4. Apply your changes and close the Declaration Editor by clicking on OK.
5. Save your work.

Step #5: Defining a Method to Update the Display

In this step you will add a method to the Color Editor interface to update the display. The `UpdateDisplay()` method calls `rgbcolor's UpdateColor()` public member function (which in turn sets the Background color of the Drawing Area). The `UpdateDisplay()` method also sets the third Label's `LabelString` value to the current RGB value. It will be invoked later, when adding behavior to the Scales.

1. Click on the Color Editor interface to select it.
2. Open the Method Editor by choosing `Selected Objects⇒Tools⇒Method Editor`.

The Method Editor appears, as shown in Figure 5-10.

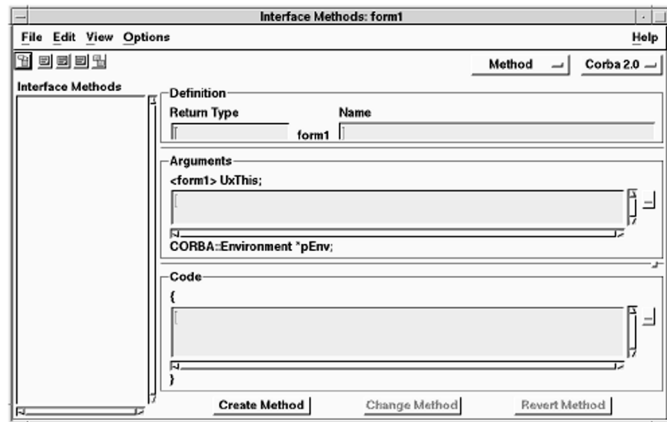


Figure 5-10 Method Editor

3. In the Return Type area, type the following:
`void`
4. In the Name area, type the following:
`UpdateDisplay`

5. In the Code area, type the following:

```
char vstring[20];  
current_color.UpdateColor(v1, v2, v3);  
sprintf(vstring, "#%02x%02x%02x", v1, v2, v3);  
UxPutLabelString( label3, vstring);
```

6. Click on Create Method. The method appears in the Interface Methods list.
7. Choose File⇒Close to close the Method Editor.
8. Save your work.

Step #6: Creating a Scale Class

In UIM/X each stand-alone interface is its own class. To create a class for a widget already embedded in an interface, you simply drag and drop the widget onto the desktop. In this step you will turn the Horizontal Scale into a class. In the next, you will enhance the class definition with new properties and behavior.

1. Click on the Horizontal Scale to select it.
2. Press and hold the Adjust mouse button, then drag and drop the Horizontal Scale onto the desktop.

A dialog appears, asking if you want to replace the Horizontal Scale just removed from the interface with an instance of it.

3. Click Yes.

UIM/X creates the component class and places an instance of it in the interface.

4. Save your work.

Step #7: Exposing Properties in the Scale Class

By default, an instance has no editable properties. However, by defining property accessor methods for the class you can make properties available in the instance. In UIM/X, creating property accessor methods is simplified using the Method Editor. Once created, the properties exposed can be used like any other.

Instances inherit all their properties from the class of which they are an instance. This allows you to create complex objects with built-in behavior then use exact replicas of them (including the behavior) in other interfaces.

For example, you could create an *About*-type Dialog Box with your company logo and standard copyright notice in it, and reuse it in all your projects without any risk of it being modified inadvertently.

To make a property available for reading or writing in an instance you create property accessor methods for the class. Property accessor methods are pairs of *get* and *set* methods. In the body of each method you specify code that operates on the desired property, usually getting or setting it. When UIM/X identifies a pair of *get* and *set* accessor methods in an instance, it presents a new property in the Property Editor.

The Method Editor provides a convenient mechanism for defining property accessor methods. Property accessor method names are of the form `ObjectName_get_MethodName` and `ObjectName_set_MethodName`. You provide the method names, while UIM/X provides the prefixes. In the body of method, you “expose” the property. In C++ Mode you can use bindings of the form `class.GetProperty` and `class.SetProperty`. In C Mode you can use the `UxGetProperty` and `UxPutProperty` functions.

Once created, the exposed property behaves like any other editable property. You can edit it by loading the instance into the Property Editor, where it shows up in the Specific category. You can provide it with a default value using the Property Editor, or set it at run time in callback code. You can also make a connection to it using the Connection Editor.

In this step you will create two pairs of *get* and *set* accessor methods for the Scale class. First you will create a pair of methods to expose the Scale’s *TitleString* property. Next you will create a pair to expose the Scale’s *Value* property. All work will be done in the Method Editor.

Creating the *TitleString* Property Accessor Methods

In this step you will create a pair of property accessor methods to expose the Scale’s *TitleString* property. In another step you will set this in the instances using the Property Editor.

To create the *TitleString* property accessor methods for the Scale:

1. Select the Scale class (the stand-alone interface), then open the Method Editor for the interface by choosing Selected Objects⇒Tools⇒Method Editor.

The Method Editor appears.

2. Change the method type from Method to `GetProperty`.

Notice the method prototype changes to reflect the required naming convention. For example, the method name prefix changes from `scaleH1to` `scaleH1_get`.

3. Edit a *get* method for the class by entering the values shown in Table 5-2

Table 5-2 Get Method Definition

In This Area	Type the Following Code
Return Type	<code>char *</code>
Name	<code>TitleString</code>
Arguments	<i>none.</i>
Code	<code>return scaleH1.GetTitleString();</code>

4. Create the new method by clicking on Create Method.

The `get` method appears in the Interface Methods area, as shown in Figure 5-11.

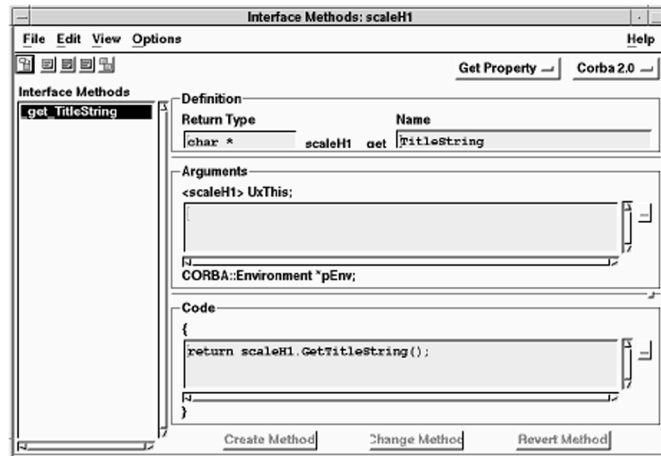


Figure 5-11 Method Editor Showing New Method

- Similarly, create a *set* method by changing to the SetProperty method function prototype.

Notice the method prototype changes once again. For convenience, the Method Editor retains much of the code you entered for the *get* method. A variable called `value` is automatically declared for the *set* method.

- Edit a *set* method for the interface by entering the values shown in Table 5-3.

Table 5-3 Set Method Definition

In This Area	Type the Following Code
Return Type	<code>void</code>
Name	<code>TitleString</code>
Arguments	<code>char *value;</code> (Be sure to change <i>int</i> to <i>char *</i> .)
Code	<code>scaleH1.SetTitleString(value);</code>

- Create the new method by clicking on Create Method. The *set* method is added to the Interface Methods area.
- Save your work.

Creating the Value Property Accessor Methods

In this step you will create a pair of property accessor methods to expose the Scale's Value property. The *get* method will return the Scale's current value. The *set* method will write the current value to the Value property.

- Change the method type to GetProperty.
- Edit a *get* method for the class by entering the values shown in Table 5-4.

Table 5-4 Get Method Definition

In This Area	Type the Following Code
Return Type	<code>int</code>
Name	<code>Value</code>
Arguments	<i>none.</i>
Code	<code>return scaleH1.GetValue();</code>

- Create the new method by clicking on Create Method.
- Similarly, create a *set* method by changing to the SetProperty method function prototype.

Enter the values shown in Table 5-5

Table 5-5 Set Method Definition

In This Area	Type the Following Code
Return Type	void
Name	Value
Arguments	int value;
Code	scaleH1.SetValue (value) ;

5. Create the new method by clicking on Create Method.
6. Save your work.

Step #8: Exposing Behavior in the Scale Class

In the last step you created methods to expose properties in an instance. Making behavior properties—normally called *callbacks*—available is equally possible. To do so you simply create a callback accessor method for the class. This is a method that follows a specific naming convention, `class.AddCallbackNameProc()`, where `class` is the widget class name, and `CallbackName` is the name you want appearing in the Property Editor. The body of the method calls `AddCallback()`, specifying the Xt callback to be used.

Since Scales return their current value during a drag and their final value when the dragging has stopped, in this step you will define two callback accessor methods for the Scale class. `DragCallback()` will return the Scale's current value. `ValueChanged()` will return the Scale's final value. All work will be performed in the Method Editor.

To create callback accessor methods for the Scale:

1. Change the method type to Method.
2. Edit a Drag callback accessor method for the class by entering the values shown in Table 5-6.

Table 5-6 Add Method Definition

In This Area	Type the Following Code
Return Type	<code>void</code>
Name	<code>AddDragCallbackProc</code>
Arguments	<code>XtCallbackProc cb;</code> <code>XtPointer client data;</code>
Code	<code>scaleH1.AddCallback(XmNdragCallback,</code> <code>cb,</code> <code>client data);</code>

3. Click on Create Method.

The `AddDragCallbackProc` callback accessor method appears in the Interface Methods area, as shown in Figure 5-12.

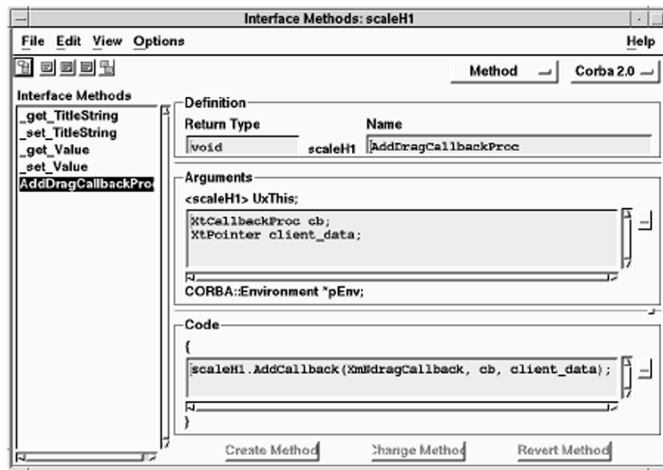


Figure 5-12 Method Editor Showing AddDragCallbackProc Method

- Similarly edit a ValueChanged callback accessor method for the class by entering the values shown in Table 5-7.

Table 5-7 Add Method Definition

In This Area	Type the Following Code
Return Type	void
Name	AddValueChangedCallbackProc
Arguments	XtCallbackProc cb; XtPointer client_data;
Code	scaleH1.AddCallback(XmNValueChangedCallback, cb, client_data);

- Create the new method by clicking on Create Method.
 The AddValueChangedProc callback accessor method appears in the Interface Methods area, as shown in Figure 5-13.

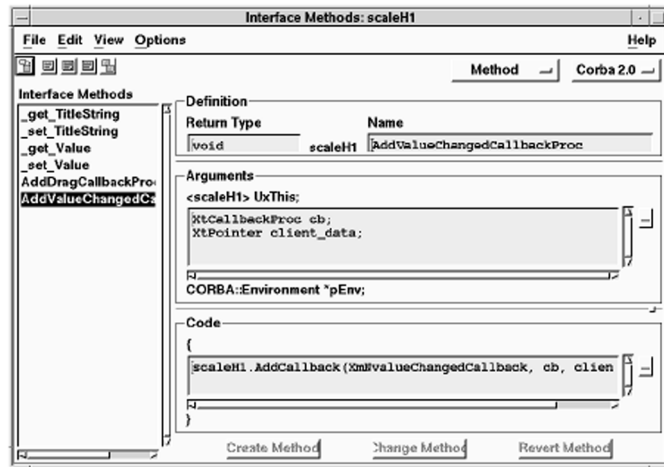


Figure 5-13 Method Editor Showing AddValueChangedProc Method

6. Close the Method Editor by choosing File⇒Close.
7. Save your work.

Step #9: Setting Properties in the Instance

In this step you will set the `TitleString` property you exposed earlier.

1. Double-click on the `scaleH1Instance1` to open the Property Editor and load the instance into it in one step.
2. Switch to the *Specific* category of Properties.

Notice the *Specific* category contains the two properties you exposed in the class, namely `TitleString` and `Value`, as shown in Figure 5-14.

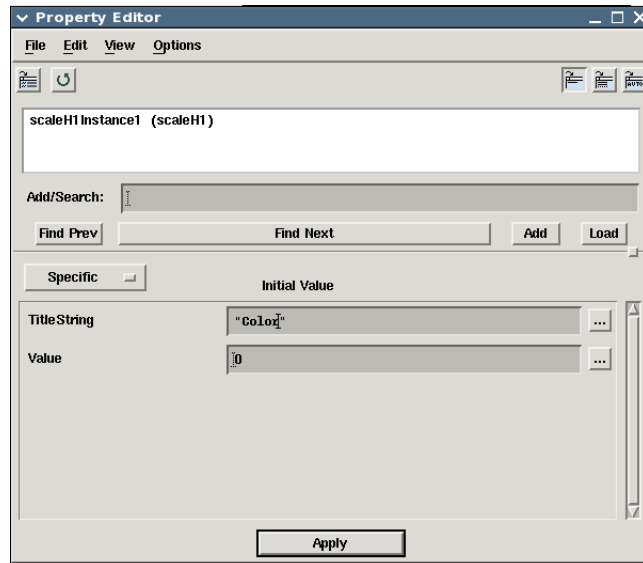


Figure 5-14 Property Editor Showing Exposed Properties

3. Change the TitleString property from "Color" to "RED".
4. Apply your changes by clicking on Apply. The interface is updated, as shown in Figure 5-15.

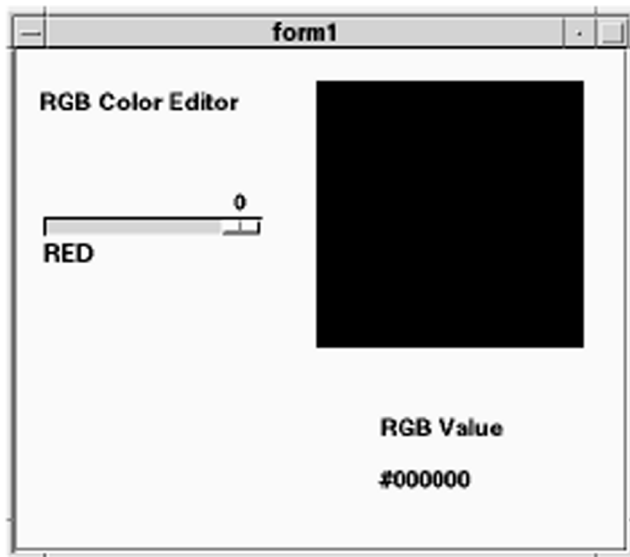


Figure 5-15 Color Editor with TitleString Property Updated

5. Close the Property Editor.
6. Save your work.

Step #10: Adding Behavior to the Instance

In this step you will add behavior to the Scale instance. First you will connect the Scale's callbacks to the Form's `UpdateDisplay` method. Next you will connect them to the Scale's own `_get_value` method, to retrieve the color component value required by `UpdateDisplay`. Finally, you will re-order the connections. All work will be performed graphically, using the Connection Editor.

Making the Scale's Connections to the Form

In this step you will load the Scale instance and the Form into the Connection Editor. Then you will connect its `DragCallback` and `ValueChangedCallback` to `UpdateDisplay`. It is important to make both connections. When a Scale stops moving its final value is returned via `ValueChanged` only. This is a Motif convention.

1. Select the RED Scale instance by clicking on it.

2. Press the Shift key and hold down the Select mouse button, then drag the cursor to the Form.

Notice a line follows the cursor. This indicates the Connection Editor is available.

3. Release the mouse button (and the Shift key) to pop up the Connection Editor, loaded with the Scale in the Source area and the Form in the Target area, as shown in Figure 5-16.

Notice the Slider's callbacks you exposed in the class, `DragCallback` and `ValueChangedCallback`, are listed in the Callback area of the Connection Editor. The method you create for the Form, `UpdateDisplay`, is listed in the Methods area.

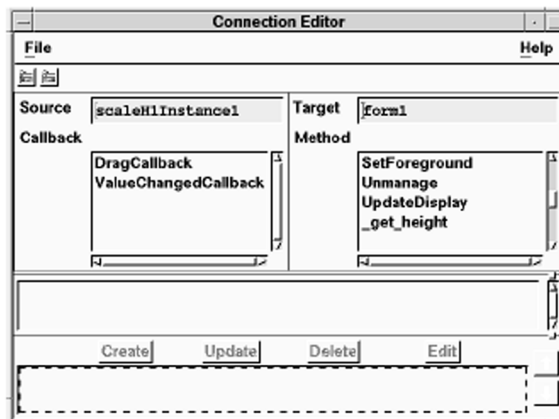


Figure 5-16 Connection Editor

4. Click on `DragCallback` in list of callbacks, and on `UpdateDisplay` in the list of methods.

The Method's default parameter appears in the Parameters list.

5. Complete the connection by clicking on `Create`.

The new connection appears in the Connection Editor, as shown in Figure 5-17.

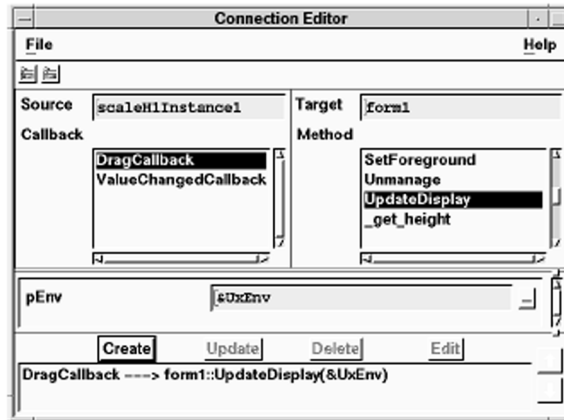


Figure 5-17 Connection Editor Showing Scale's First Connection

6. Click on ValueChangedCallback in list of callbacks (keeping UpdateDisplay highlighted in the Methods list).
7. Complete the connection by clicking on Create. Again, the display is updated to show the new connection.
8. Save your work.

Making the Scale Instance's Connections to Itself

Recall that UpdateDisplay calls rgbcolor's UpdateColor public member function, passing in three values, one each for the red, green, and blue components of a color. The function is shown below, for reference:

```
char vstring[20];
current_color.UpdateColor(v1, v2, v3);
sprintf(vstring, "#%02x%02x%02x", v1, v2, v3);
UxPutLabelString( label3, vstring);
```

In this step you will retrieve a value for v1 from the Scale instance's own `_get_Value` method that you created earlier to expose its Value property. As before, you will connect both the Dragcallback and ValueChangedcallback to `_get_Value`.

1. Load the Scale instance itself into the Target area of the Connection Editor:
 - Select the Scale instance, `scaleH1Instance1`, then click on the Load Target icon (the right-most one).
 - Or, drag and drop the Scale instance into the Target area.

The instance's methods are displayed in the Methods list, including those you defined for it, `_get_Value`, and `_set_Value`.

2. Click on `DragCallback` in the Callback list, and `_get_Value` in the Method list.
3. For the Return parameter, enter the following value:

`v1`

The variable `v1` will hold the red RGB value.

4. Complete the connection by clicking on Create.

The new connection appears in the Connection Editor, as shown in Figure 5-18.

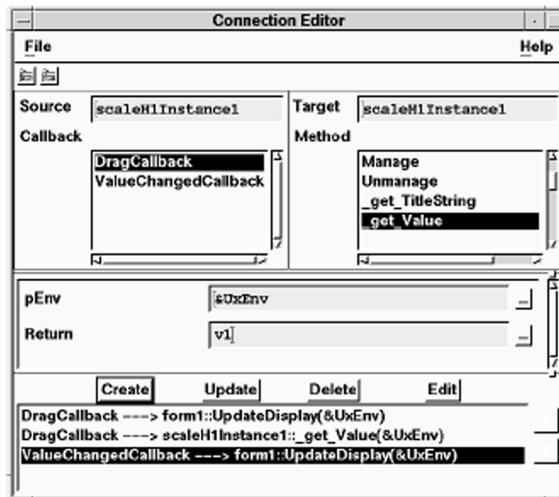


Figure 5-18 Connection Editor Showing Three or Four Connections for Scale

5. Next, click on `ValueChangedCallback` in the Callback list (keeping `_get_Value` highlighted in the Methods list).
6. Ensure the Return parameter has the following value:
 - `v1`
7. Complete the Connection by clicking on Create.

The Connection Editor now shows four connections for the Scale instance, as shown in Figure 5-19.

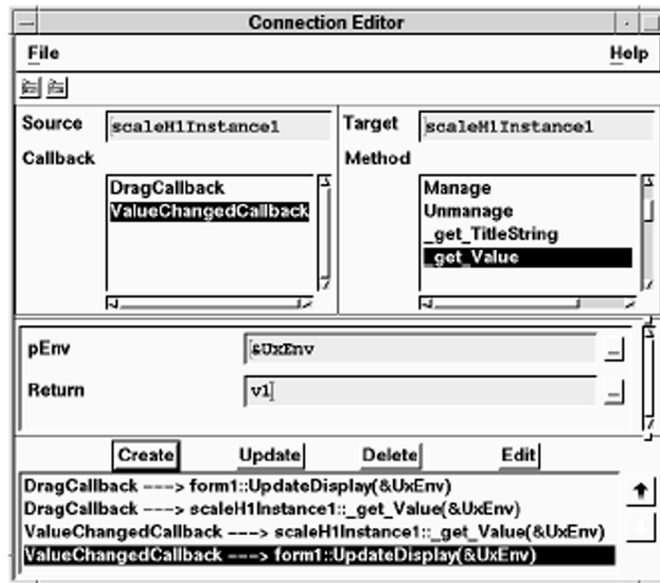


Figure 5-19 Connection Editor Showing All Four Connections for Scale

8. Save your work.

Reordering the Connections

Connection event-action pairs are executed in the order in which they appear in the Connection Editor. In this step you will reorder the connections so `UpdateDisplay` receives the most recent value from the Scale.

1. Highlight the following line in the connections list

```
DragCallback --->
scaleH1Instance1::_get_Value (&UxEnv)
```

2. Click on the up-arrow to move the `DragCallback::get_Value` connection to the top of the list.

3. In the same way, highlight the following connection:

```
ValueChangedCallback --->
scaleH1Instance1::_get_Value (&UxEnv)
```

4. Move it until it is just above the `ValueChangedCallbackUpdateDisplay` connection.

When complete the connections should appear in the following order:

```
DragCallback --->
scaleH1Instance1::_get_Value (&UxEnv)

DragCallback ---> form1::UpdateDisplay (&UxEnv)

ValueChangedCallback --->
scaleH1Instance1::_get_Value (&UxEnv)

ValueChangedCallback --->
form1::UpdateDisplay (&UxEnv)
```

5. Close the Connection Editor by choosing `File⇒Close`.
6. Save your work.

Step #11: Completing the Interface

With the `Scale` class defined and behavior added you are now ready to complete the interface. First you will duplicate the `Scale` instance and update its `TitleString` property. Next you will update the connections for the duplicate `Scales`, setting their `Return` values to write to the global variables `v2` and `v3`, the green and blue components of a color.

Duplicating the Scale and Updating Properties

In this step you will duplicate the `Scale` instance and set the `TitleString` properties.

1. Click on the `Scale` instance to select it then choose `Selected Objects⇒Duplicate`.
2. Position the new instance under the first.
3. Duplicate the `Scale` instance once again, positioning the third instance under the first two.

The interface should now appear as shown in Figure 5-20.

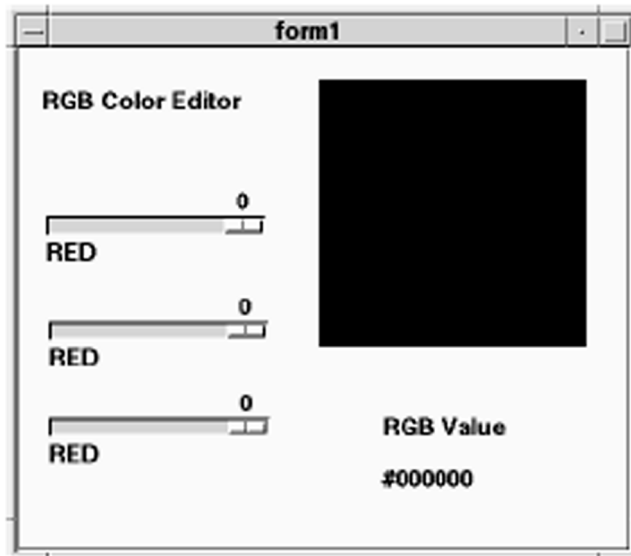


Figure 5-20 Color Editor with All Three Scale Instances

4. Double-click on the second Scale instance, `scaleH1Instance2` to load it into the Property Editor.
5. Locate the `TitleString` property in the `Specific` category, changing it from "RED" to "GREEN".
6. Apply your changes.
7. Similarly, load the third Scale instance, `scaleH1Instance3` into the already open Property Editor by dragging and dropping, or selecting it and clicking on the Load icon.
8. In this case, change the `TitleString` property from "RED" to "BLUE" and apply your changes.

The interface should now appear as shown in Figure 5-21.

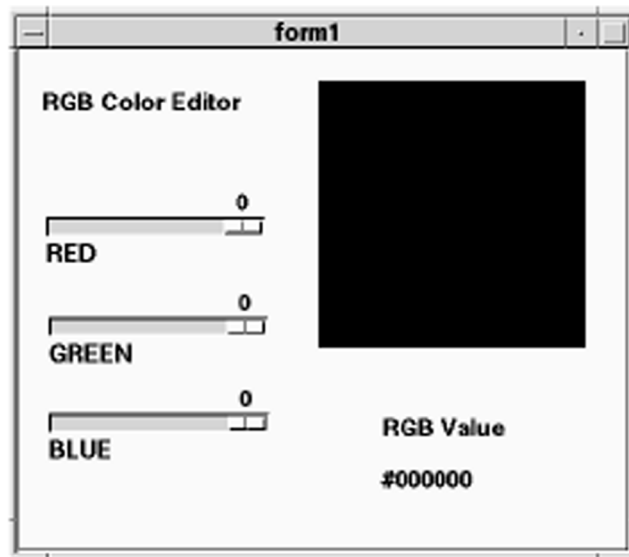


Figure 5-21 Color Editor with Scale TitleString Properties Updated

Updating the Connections

In this step you will set the Return values for the duplicate scales to write to the global variables `v2` and `v3`, the green and blue components of a color.

1. Load the second Scale instance, `scaleH1Instance2`, into the Connection Editor by selecting it and choosing Selected Objects⇒Tools⇒Connection Editor.
2. Highlight the following connection in the Connections list:


```
DragCallback --->
scaleH1Instance2::_get_Value(&UxEnv)
```
3. Click on Edit.

Notice the Scale instance is loaded into the Target area, and its `_get_Value` method is automatically highlighted.
4. Change the Return parameter from `v1` to `v2`. `v2` is the variable used to set the green value.
5. Update the connection by clicking on Update.

- Similarly, begin updating `ValueChangedCallback` by highlighting the following connection in the Connections list:

```
ValueChangedCallback --->
scaleH1Instance2::_get_Value (&UxEnv)
```

- Click on Edit.
- Change the Return parameter from `v1` to `v2`, and click on Update.
- Repeat the process for the BLUE scale instance, `scaleH1Instance3`, changing the Return parameters from `v1` to `v3`.
- When complete, close the Connection Editor by choosing `File⇒Close`.
- Save your work. The interface is now ready for testing.

Step #12: Testing the Program

Before generating code for the project in the next section, take a moment to switch to Test Mode.

- Switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

- Hide the Color Editor interface by choosing `View⇒Hide Project`.
- Choose `Tools⇒Interpreter`. The Interpreter window appears, as shown in Figure 5-22.

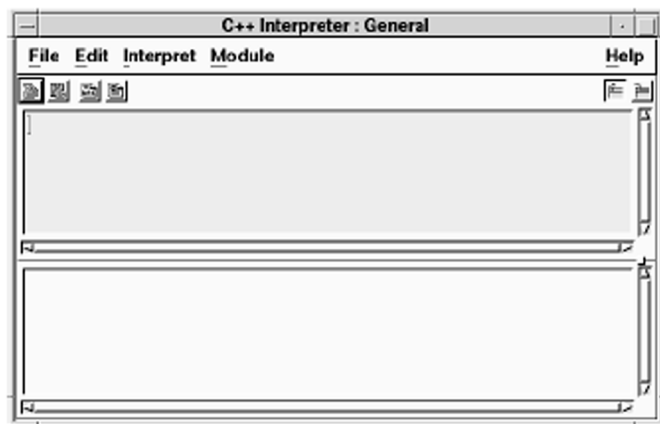


Figure 5-22 C++ Interpreter Window

- Select the Form interface by clicking on its icon in the Project Window.

5. Choose Module⇒Selected Interface in the Interpreter or click on the corresponding icon .

The Interpreter title bar changes to reflect the new scope.

Enter the following code in the Interpreter window, as shown in Figure 5-23.

```
UxPopupInterface (create_form1 (UxParent) ,
no_grab) ;
```

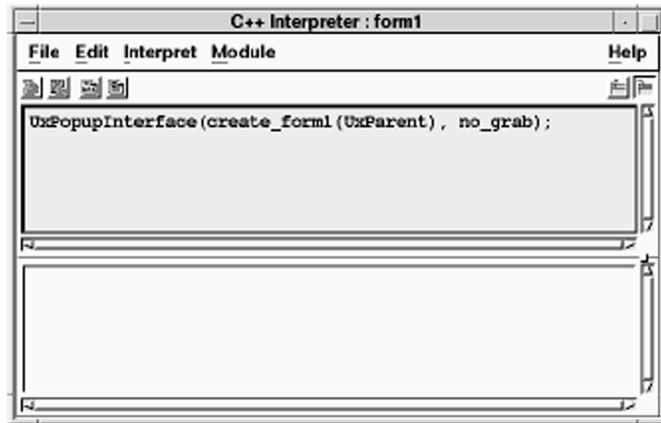


Figure 5-23 Evaluating the Interface's Create Function

6. Triple-click the line of code to highlight it, then choose Interpret⇒Evaluate.

The Interpreter evaluates the code, pops up the Color Editor interface, and prints the following to the Interpreter Messages Area:

```
Result : 0
```

7. Close the Interpreter by choosing File⇒Close.
8. Test the Color Editor interface:
 Sliding a Scale updates the value on the scale, the value in the Label, and the Drawing Area widget itself.
9. When you are through, switch back to Design Mode by clicking on the Design icon.

Step #13: Generating the C++ Code and Running the Executable

You have now successfully constructed and tested the Command Line project. In this step you will generate the code for the application, and run it, without leaving the development environment.

1. Check that you are in Design Mode. If not, click on the Design icon.
2. Choose Options⇒Code Generation on the Project Window menu. The Code Generation Options window appears, as shown in Figure 5-24.

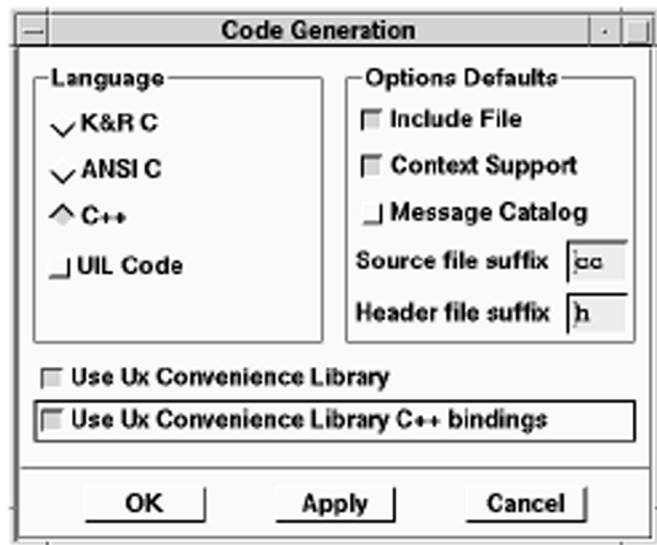


Figure 5-24 Code Generation Options

3. Ensure that the following radio buttons and toggle buttons are selected:
 - C++
 - Context Support

Context support is required when generating code for an interface that uses instances.

4. Save your changes and close the dialog by clicking on OK.
5. Click on the Run Mode icon or choose File⇒Generate Code As... on the Project Window menu.

6. Check that the following radio buttons and toggle buttons are selected:
 - Write All Interfaces
 - Run Makefile
 - Write Main Program
 - Run Executable
 - Write Makefile
7. Click OK to generate your code.

UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
8. Test your program. Verify that it works as it did in Test Mode.
9. To stop the program select Close from the window control box, or switch back to Design Mode by clicking on the Design icon .
10. Save the changes to your program.

Now when you modify the Color Editor project, you can simply click on the Run Mode icon to generate the code, compile it, and run the executable in one step.

Integrating a Non-Visual Object

6

Overview

UIM/X provides the objects and tools needed to develop sophisticated applications with graphical user interfaces quickly and easily. But some objects, by their very nature, cannot be represented visually. Files, servers, database objects and data structures, for example, have no graphical user interface. Yet application developers and development teams would benefit greatly from these and other non-visual objects. With the Non-Visual Shell, UIM/X provides the structure for developing non-visual object classes. By integrating them into UIM/X you can extend the UIM/X palette to include your new non-visual object classes, facilitating their use by you and your design team.

Consider the advantages of a non-visual linked list object, for example. Instead of declaring the linked list in the Declaration Editor you could simply drop an instance of a linked list *object* from the palette into the interface where you need it. If the linked list were equipped with *add* and *delete* methods you could call these from your interface callback code, or connect to them visually using the Connection Editor.

As the above example illustrates, the Non-Visual Shell object extends the benefits of the UIM/X development environment to the non-visual aspects of your application. Using the Non-Visual Shell you can graphically create any non-visual object you need. Then you can use the Method Editor to give it functionality.

Integrating the non-visual object into the UIM/X development environment provides additional advantages. As with a visual GUI object, you can pre-register its *create* function and methods with the interpreter for faster processing. Further, by placing an instance of the object into the palette, you can ensure that its methods are available for use, while remaining uneditable.

About This Tutorial

This tutorial demonstrates how to create a non-visual object and integrate it into UIM/X. While you will create the new object using the Non-Visual Shell, to facilitate using it in an application, a start-up project has been provided. As part of the development process you will test the new object by using it in the start-up project. Once integrated into the UIM/X executable, you will use the non-visual object in the start-up project once again, this time by dragging and dropping the integrated object from the Palette.

Note: If you have installed UIM/X in its C-only configuration, do not attempt the tutorial in this chapter. The tutorial assumes that UIM/X is running in C++ mode, and that you have a C++ compiler.

The GUI You Will Build

In this chapter you will create a non-visual object to open, read, write, and close files. You will then use the new object in a To Do List application in two ways. First, you will use it directly, by simply adding an instance of the object to the To Do List main interface. Next, you will augment the UIM/X executable with the new object, and use the integrated version in the same To Do List application.

To allow the tutorial to focus on development of the File object and augmenting UIM/X, the To Do List application has been provided as a start-up project, as shown in Figure 6-1.

The interface consists of the following areas:

- *Menu Bar:* Contains pull-down menus with commands to open and close a file, and exit the application. The Help menu pops up a Message Box dialog.
- *Work Area:* Contains a Scrolled Window where you can view, edit, or delete the tasks in your To Do List.
- *Push Button Area:* Contains a Field for editing tasks, a slide bar for setting the task's priority, and Push Buttons for adding the task to the work area, or clearing the Field.

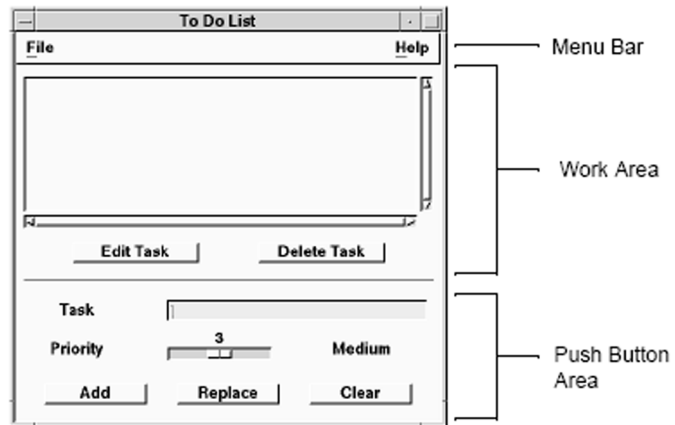


Figure 6-1 To Do List

The Sections in This Tutorial

This tutorial takes about 120 minutes to complete. It contains the following sections:

- Section I: Creating a Non-Visual File Object
- Section II: Using the File Object in the To Do List
- Section III: Integrating the File Object into UIM/X
- Section IV: Using the Integrated File Object

Section I: Creating a Non-Visual *File* Object

This section focuses on creating the non-visual object with the following features:

- The object will contain all the code it needs to open, read, write, and close a file.
- Methods will be used so other interfaces have access to the object's functions.

In this section you will start UIM/X in Standard Mode. Next you will create the non-visual File object based a Non-Visual Shell, modifying its create function to allow the File object to receive a file name. Finally, you will add methods to the File object to open, close, read, and write files. These methods will be used in the next section, when you use the File object in a To Do List project.

The Steps in This Section

This section takes about 20 minutes to complete. It contains the following steps:

Step #1: Starting UIM/X in Standard Mode

Step #2: Creating the Non-Visual File Object

Step #3: Adding Functionality to the File Object

Where You Are in the Tutorial

⇒Section I: Creating a Non-Visual File Object

Section II: Using the File Object in the To Do List

Section III: Integrating the File Object into UIM/X

Section IV: Using the Integrated File Object

Step #1: Starting UIM/X in Standard Mode

Before you begin building the File object, set up new directories and copy the start-up project as follows:

1. Start the X Window System.
2. Bring up a terminal window.
3. Make a base directory for this tutorial:

```
mkdir chap6
```

4. Change to the directory you just created:

```
cd chap6
```

5. Make a directory to store the files you will create in this section:

```
mkdir sect1
```

6. Change to the directory you just created:

```
cd sect1
```

7. Start UIM/X from your new directory:

```
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 &
```

After a brief pause, a copyright notice window appears, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and UIM/X palette appear.

8. Iconify the terminal window.

Note: To restart the tutorial, begin again from Step 4 above.

Step #2: Creating the Non-Visual *File* Object

UIM/X provides a Non-Visual Shell class especially designed for creating nonvisual objects. By modifying the new object's create function, you can easily pass values to it. You can also declare variables for use with the object's methods.

Adding a parameter to an object's create function allows you to pass values to it each time it is created. The parameter also shows up in the Property Editor for instances of the object, so you can give it a default value, or assign it a value at runtime. In this case, the parameter will be used to pass the name of the file to be manipulated.

In this step you will use a Non-Visual Shell widget to create the File object. Next you will use the Declaration Editor to define a few global variables to function as error messages. You will also modify its *create* function to add a *filename* parameter. The error messages and *filename* parameter will be used in the next step, when you create methods for the File object to read, write, open and close files.

Drawing the *File Object*

In this step you will create the File object, based on the Non-Visual Shell.

1. Drag and draw (or drag and drop) a Non-Visual Shell from the Shells area of the Palette to your work area, as shown in Figure 6-2.

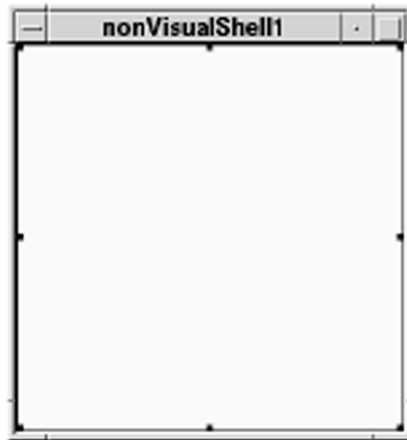


Figure 6-2 Non-Visual Shell Added to the Project

It does not matter what size you make the Non-Visual Shell.

2. Open the Property Editor by double-clicking on the Non-Visual Shell.
The property Editor appears, loaded with the Non-Visual Shell, as shown in Figure 6-3.

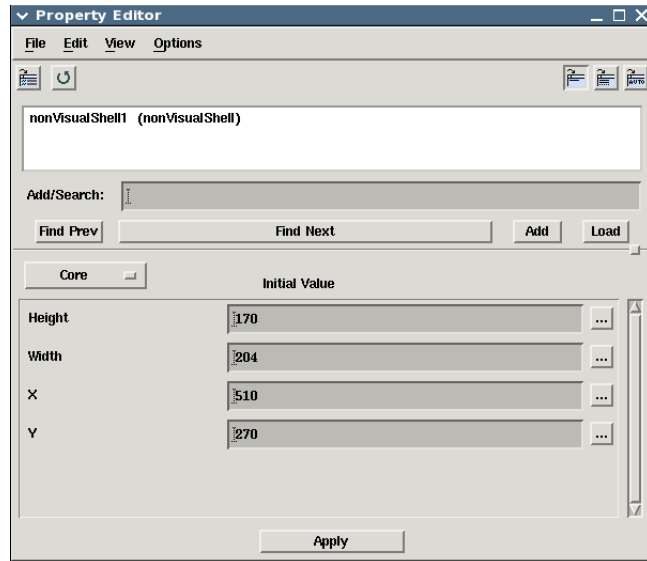


Figure 6-3 Property Editor Loaded with Non-Visual Shell

Note that while the Non-Visual Shell has dimension and position properties, these apply to Design Mode and Test Mode only. At runtime Non-Visual Shells, as the name implies, are not visible.

3. In the Declaration category, locate the Name property, and change it from `nonVisualShell1` to `File`.
4. Apply the change by clicking on Apply.
5. Close the Property Editor by choosing `File⇒Close` in the Property Editor.
6. Save the interface as a project by choosing `File⇒Save Project As` in the Project Window.
7. Check that the project name selection box shows the path to your work directory, `chap6/sect1`.
8. Click in the selection box and replace `Untitled.prj` with `file.prj`, then click OK to save your project.

Defining Global Variables for the *File* Object

In this step you will load the File object into the Declaration Editor and define a few global constants. These will be shared by the object's methods that you will add later. You will also add a *filename* parameter to the File object's *create* function. The parameter will be used to name the file operated upon by the File object.

1. Open the Declaration Editor by pressing the Menu mouse button while over the interface, and choosing Selected Objects⇒Tools⇒Declaration Editor.
2. The Declaration Editor appears, as shown in Figure 6-4.

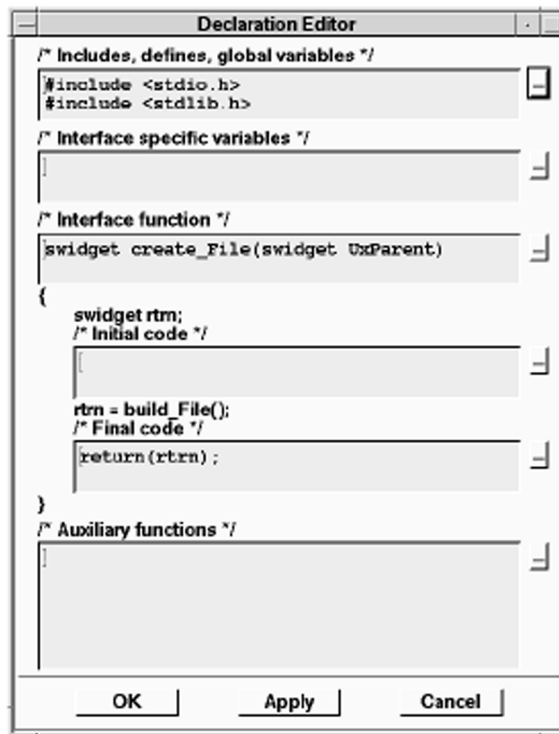


Figure 6-4 Declaration Editor

- Click on the editor button (...) next to the `/* Includes, defines, global variables */` area, and add the following macro constants:

```
#define FILE_NOERROR 1
#define FILE_ERROR_ALREADY_OPEN -1
#define FILE_ERROR_CANNOT_OPEN -2
#define FILE_ERROR_CANNOT_CLOSE -3
#define FILE_ERROR_NOT_OPEN -4
#define FILE_ERROR_EOF -5
```

- Close the Text Editor by clicking on OK.
- Add the following definition to the `/* Interface specific variables */` area:

```
FILE *file_ptr;
```

- Open a text editor for the interface function area by clicking on the editor button (...) next to the `/* Interface function */` area.

The interface function appears as follows:

```
swidget create_File(swidget UxParent)
```

- Add a parameter, `filename`, to the interface function:

```
swidget create_File(swidget UxParent, char*
    filename)
```

The *filename* parameter will be used to hold the name of the file opened by the object's methods.

- Close the Text Editor by clicking on OK.
- Add the following code to the `/* Final Code */` area, just before the return call (the return call is shown below, for reference).

```
file_ptr = NULL;
return (rtrn);
```

- Apply the changes by clicking on OK in the Declaration Editor.
- Save your work.

Step #3: Adding Functionality to the File Object

Now that you have created a File object that is not visible at runtime, the next step is to add functionality to it. Methods present the most convenient way to add functionality to objects designed for integration, for two reasons. First, methods are inherited by instances of the object, with an automatic naming convention that makes them easy to access. Second, when augmenting UIM/X with a new object, methods provide functionality while remaining uneditable.

When you place an instance of an object into an interface, any methods you defined for it become available for use. They are visible in the Connection Editor, and are available in callback code. But the bodies of an object's methods are not editable in the instance, providing structured access, and security for the underlying code.

In UIM/X method names are of the form `InterfaceName_MethodName`. The first part of the name reflects the interface (or object) containing the method, and is automatically provided by UIM/X. The second part you provide when creating or editing the method.

In this step you will use the Method Editor to add four methods to the File object. The methods will open, close, read and write a file respectively, with error-checking. In Section II: Using the File Object in the To Do List, you will add an instance of the File object to the To do List interface and call the methods.

Note: Since the results of file operations are not directly observable, this step makes use of `#ifdef DESIGN_TIME` statements to write messages to the Project Window when in Test Mode. `DESIGN_TIME` is a macro constant defined when in Design Mode and Test Mode but not at runtime.

Creating the Open Method

In this step you will add a method to the File object for opening files. In Test Mode the open method will write to the Messages area of the Project Window as well.

1. Click on the File interface to select it.
2. Open the Method Editor by choosing Selected Objects⇒Tools⇒Method Editor.
3. The Method Editor appears, as shown in Figure 6-5.

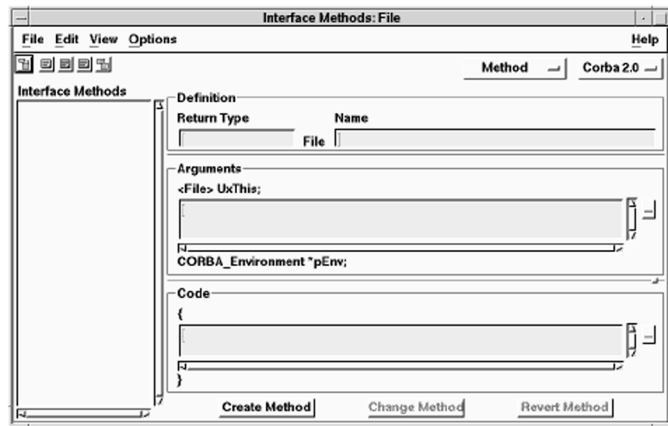


Figure 6-5 Method Editor

4. In the Return Type area, type the following:
`int`
5. In the Name area, type the following:
`Open`
6. In the Arguments area, type the following:
`char *access;`

7. In the Code area, type the following:

```
#ifdef DESIGN_TIME
printf("Trying to open %s \n", filename);
#endif /* DESIGN_TIME */

if (file_ptr != NULL)
{
#ifdef DESIGN_TIME
printf("%s is already open\n", filename);
#endif /* DESIGN_TIME
*/return FILE_ERROR_ALREADY_OPEN;
}
file_ptr = fopen( filename, access );

if (file_ptr == NULL)
{
#ifdef DESIGN_TIME
printf("%s cannot be opened\n", filename);
#endif /* DESIGN_TIME */
return FILE_ERROR_CANNOT_OPEN;}
#ifdef DESIGN_TIME
printf("%s opened\n", filename);
#endif /* DESIGN_TIME */
return FILE_NOERROR;
```

8. Click on Create Method. The method appears in the Interface Methods list.
9. Save your work.

Creating the Remaining Methods

In this step you will create the methods to close, read, and write a file.

1. Repeat the process to create the *Close* method.

The complete definition for the *Close* method is shown in Table 6-1.

Table 6-1 Close Method Definition

Area	Code
Return Type	int
Name	Close
Arguments	<i>none.</i>
Code	<pre> #ifdef DESIGN_TIME printf("Trying to close %s \n", filename); #endif /* DESIGN_TIME */ if (file_ptr == NULL) { #ifdef DESIGN_TIME printf("%s not open\n", filename); #endif /* DESIGN_TIME */ return FILE_ERROR_NOT_OPEN; } if (fclose(file_ptr) == EOF) { #ifdef DESIGN_TIME printf("Unable to close %s \n", filename); #endif /* DESIGN_TIME */ return FILE_ERROR_CANNOT_CLOSE; } #ifdef DESIGN_TIME printf("%s closed\n", filename); #endif /* DESIGN_TIME */ file_ptr = NULL; return FILE_NOERROR; </pre>

2. Once you have entered the code for the *Close* method, be sure to click on Create to define the method.
3. Repeat the process for the *Readline* method, as shown in Table 6-2.

Table 6-2 Readline Method Definition

Area	Code
Return Type	char *
Name	Readline
Arguments	none.
Code	<pre>static char str[256]; #ifdef DESIGN_TIME printf("Trying to read from %s \n", filename); #endif /* DESIGN_TIME */ if (file_ptr == NULL) { #ifdef DESIGN_TIME printf("%s is not open\n", filename); #endif /* DESIGN_TIME */ return NULL; } if (fgets(str, sizeof(str), file_ptr) == NULL) { #ifdef DESIGN_TIME printf("no more in %s\n", filename); #endif /* DESIGN_TIME */ return NULL; } #ifdef DESIGN_TIME printf("read from %s \n", filename); #endif /* DESIGN_TIME */ return str;</pre>

- Repeat the process one last time for the *Writeline* method, as shown in Table 6-3.

Table 6-3 Writeline Method Definition

Area	Code
Return Type	int
Name	Writeline
Arguments	char *line;
Code	<pre> #ifdef DESIGN_TIME printf("Trying to write to %s \n", filename); #endif /* DESIGN_TIME */ if (file_ptr == NULL) { #ifdef DESIGN_TIME printf("%s is not open\n", filename); #endif /* DESIGN_TIME */ return FILE_ERROR_NOT_OPEN; } #ifdef DESIGN_TIME printf("%s written to\n", filename); #endif /* DESIGN_TIME */ fprintf(file_ptr, "%s\n", line); return FILE_NOERROR; </pre>

5. Choose File⇒Close to close the Method Editor.
6. Save your work. The File object is now complete.

Section II: Using the File Object in the To Do List

Now that you have finished creating the non-visual File object, you can put it to use in a project. Since you defined the object's functionality using methods, the simplest way to use it is to add an *instance* of it to the interface. Adding an instance makes the object's methods available for use in the interface, while keeping them local to the interface. Values can be passed to and from the instance directly, without the need for declaring them as *extern*.

In this section you will load the To Do List start-up project provided. Next you will add an instance of the File object to the To Do List. Then you will modify the menus to work using the File object's methods. After testing the To Do List, you will generate code for the project.

This section takes about 20 minutes to complete. It contains the following steps:

- Step #4: Loading the Start-Up Project
- Step #5: Adding an Instance of the File Object to the Interface
- Step #6: Modifying the To Do List Menus
- Step #7: Testing the To Do List
- Step #8: Generating the Code and Running the Executable

Where You Are in the Tutorial

- Section I: Creating a Non-Visual File Object
- ⇒Section II: Using the File Object in the To Do List
- Section III: Integrating the File Object into UIM/X
- Section IV: Using the Integrated File Object

Step #4: Loading the Start-Up Project

To facilitate development of the To Do List, a start-up project has been provided. It contains the To Do List main interface with menus already defined, plus a Message Box. In this step you will load the start-up project.

To Load the Start-Up Project

1. Under the `chap6` directory, make a directory to store the files you will create in this section of the tutorial:


```
mkdir sect2
```
2. Change to the directory you just created:


```
cd sect2
```
3. Copy the To Do List project files into your work directory:


```
cp uimx_directory/contrib/ToDoList/* .
```
4. Change the permissions on the project files you copied to make them writable:


```
chmod a+w *
```
5. In the same way, copy the File interface from the `chap6/sect1` directory into your current directory:


```
cp ../sect1/File.i .
```
6. Reset UIM/X by choosing File⇒Reset in the Project Window. Before loading a new project, it is helpful to reset UIM/X.

7. Choose File⇒Open in the Project Window.
8. Navigate to chap6/sect2, choose ToDo.prj, and click OK.
Dialogs will appear indicating that you are loading an interface originally created in Novice Mode, and that compound objects will be converted to widgets.
9. Dismiss each dialog as it appears by clicking Replace. The To Do List start-up interface appears, as shown in Figure 6-6.

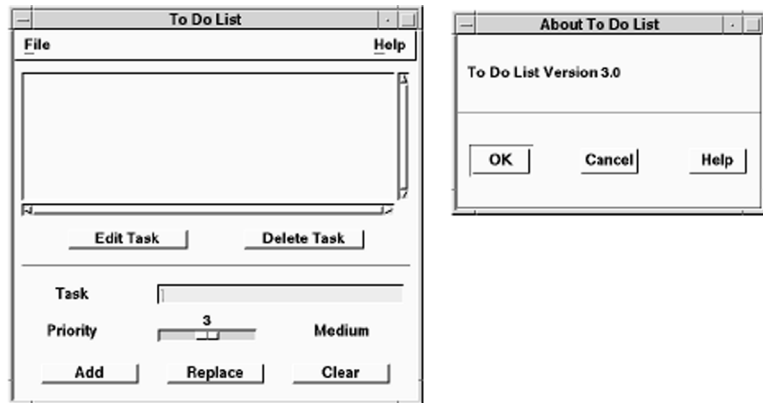


Figure 6-6 To Do List Start-Up Project

The Message Box is not visible by default, though an icon appears for it in the Project Window. This interface is popped up by a callbacks provided in the menus. You will test it later in this section.

Step #5: Adding an Instance of the File Object to the Interface

Now that you have loaded the To Do List interface, you can add an instance of the File object to it. Placing an instance on the interface allows you to refer to its methods directly, without declaring them as external to the interface.

1. Add the File object to the To Do List project by choosing File⇒Open in the Project Window, and selecting `File.i`.
2. Click on the File interface to select it. An interface must be selected to create an instance of it.
3. Point to the To Do List interface, then press and hold the Menu mouse button to display the Selected Objects popup menu.

Notice the selection “Instance of File” appears on the menu as shown in Figure 6-7.

Selected Objects (appform1)	
Tools	/
Cu<u>t</u>	Ctrl+X
Co<u>py</u>	Ctrl+C
<u>P</u> aste	Ctrl+V
Duplicate	
Al<u>ig</u>n	/
Ar<u>ra</u>nge	/
De<u>le</u>te	
Other	/
Managers	/
Primitives	/
Gadgets	/
Menus	/
Instance of File	

Figure 6-7 Selected Objects Popup Menu

4. Choose “Instance of File”. The cursor changes into the corner shape.

5. Drag and draw the instance of the File on the To Do List interface.
Make sure to add the instance to the work area of the interface. It does not matter what size you draw it, since the instance will not be visible. You will work with the instance via the Browser.
6. Open the Browser by choosing Selected Objects⇒Tools⇒Browser while over the To Do List interface.

The Browser appears, as shown in Figure 6-8.

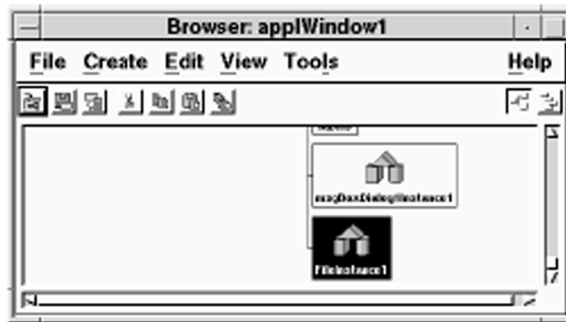


Figure 6-8 Browser Showing the Instance of the File Object,
FileInstance1

7. Load the File instance, `FileInstance1`, into the Property Editor by selecting it in the Browser and choosing Selected Objects⇒Tools⇒Property Editor.
8. In the Core category locate the `filename` property, changing it from `NULL` to `"todo.out"`.
The `filename` property is the result of having added a parameter to the File object's create function. This property specifies the file that will be opened and closed.
9. In the Declaration category locate the `Name` property, changing it from `FileInstance1` to `todoFile`.
10. Apply your changes by clicking on Apply in the Property Editor.
11. Close the Property Editor by choosing File⇒Close.
12. Save your work.

Step #6: Modifying the To Do List Menus

The menus provided with the To Do List project already contain functionality to open and close files. In this step you will change the callbacks to use the methods defined for the File object instead.

1. Select the menu bar and open the Menu Editor by choosing Tools⇒Menu Editor.

The Menu Editor appears, as shown in Figure 6-9.

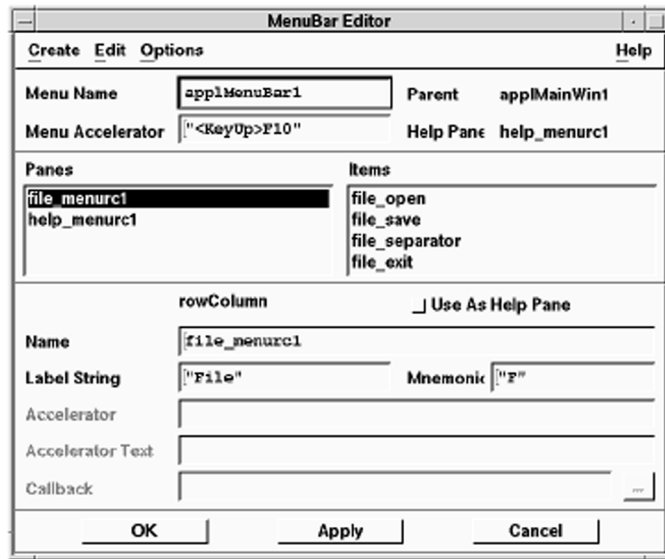


Figure 6-9 Menu Editor

2. Select the `file_open` item. The properties and callbacks are listed in the display area.
3. Click on the Text Editor button (...) beside the `Callback` property. A Text Editor appears, displaying the code included with the To Do List project.

4. Delete the code provided, replacing it with the following code:

```
char *str;
    Widget w;
    XmString xms;

if ( File_Open(todofile, "r", &UxEnv) )

{
w = UxGetWidget(scrolledList1);
XmListDeleteAllItems(w);
while ( str = File_Readline(todofile, &UxEnv) )
{
/* remove trailing newline and add to the list */
str[strlen(str)-1] = '\0';
xms = XmStringCreateSimple(str);
XmListAddItem(w, xms, 0);XmStringFree(xms);
}
    File_Close(todofile, &UxEnv);
}
```

5. Close the Text Editor and copy the code to the callback by clicking OK.
6. Apply the changes without closing the Menu Editor by clicking Apply.

7. Similarly, enter the following callback for the `file_save` item.

```
char *taskList;
    char *processList;

    if ( File_Open(todofile, "w", &UxEnv) )
    {
        taskList = UxGetItems(scrolledList1);
        if (taskList)
        {
            /* Replace commas by newlines and write out to the
            list */
            processList = taskList;
            while (*processList)
            {
                if (*processList == ',')
                {
                    *processList = '\\n';
                }
                processList++;
            }
            File_Writeline(todofile, taskList, &UxEnv);
            File_Close(todofile, &UxEnv);
        }
    }
```

8. Apply the changes and close the Menu Editor by clicking OK.
9. Save your work.

Step #7: Testing the To Do List

Before generating code for the project in the next section, take a moment to test the menus.

1. Switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

2. Test the To Do List interface:
 - To add a task to the work area, type your text in the Task field, and click on Add.
 - Use the slider to assign the task a priority level.
 - To edit an existing task, highlight it in the work area, and click on Edit. The task is copied to the Task area.
3. Test the File functionality:
 - Choosing File⇒Save writes the contents of your to do list to `todo.out`.
 - Choosing File⇒Open displays the contents of `todo.out` in the work area.
 - Note that since `DESIGN_TIME` is defined in Test Mode, appropriate messages are printed to the Project Window.
4. When you are through, switch back to Design Mode by clicking on the Design icon.

Step #8: Generating the Code and Running the Executable

The final step in this section is to generate code for the To Do List project.

1. Check that you are in Design Mode.
2. Choose File⇒Generate Project Code As on the Project Window menu.

3. Check that the following radio buttons and toggle buttons are selected:
 - Write All Interfaces
 - Run Makefile
 - Write Main Program
 - Run Executable
 - Write Makefile
4. Click OK to generate your code.

UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.
5. Test your program. Verify that it works as it did in Test Mode (minus the status messages).
6. To stop the program choose File⇒Exit.
7. Save the changes to your program.

Now when you modify the To Do List project, you can simply click on the Run Mode toggle to generate the code, compile it, and run the executable in one step.
8. Since you will restart UIM/X in the next section, exit now by choosing File⇒Exit in the Project Window.

Section III: Integrating the File Object into UIM/X

Fundamental to integrating a new object into UIM/X is creating a new class definition for the object. The class definition contains information about the new object that allows it to be added to the UIM/X executable and used like any other widget. It contains code that lets you use it in projects where both C or C++ code may be generated. It can also provide a common API and user interface to the component, and can determine what properties and callbacks are available for it in the Property Editor.

UIM/X has been designed for easy integration of the custom objects you create. First you generate a C++ class definition for the object. This code includes additional “wrapper” and “integration” code required by UIM/X. For an object created within UIM/X, this code can be produced automatically.

While generating class code in UIM/X is automatic, it requires options not normally available in the Code Generation Options window. To make these options visible, you must set Builder Engine resources and restart UIM/X.

Once you recompile UIM/X with the class definition, the new object becomes available for use, and is indistinguishable from those provided with UIM/X. You can drag and drop it from a palette, and make use of its methods from within your callback code. In addition, Test Mode, run mode, and code generation work just as expected.

In this section you will augment the UIM/X executable with the File object created earlier. You will begin by restarting UIM/X, this time with Builder Engine resources set. Next you will generate the class code for the File object. Then you will compile the class code twice—once for use with the UIM/X executable itself, and once for use with any project code generated using UIM/X. Once added to the executable, you will add the File object to the palette, and “polish” the augmented UIM/X to load the palette at start up.

This section takes about 60 minutes to complete. It contains the following steps:

Step #9: Restarting UIM/X with Builder Engine Resources

Step #10: Creating the New Class Code

Step #11: Compiling the New UIM/X Class Code

Step #12: Augmenting UIM/X

Step #13: Creating a New UIM/X Palette

Step #14: Polishing the Augmented UIM/X

Where You Are in the Tutorial

Section I: Creating a Non-Visual File Object

Section II: Using the File Object in the To Do List

⇒Section III: Integrating the File Object into UIM/X

Section IV: Using the Integrated File Object

Step #9: Restarting UIM/X with Builder Engine Resources

UIM/X users are accustomed to setting code generation options prior to generating project code. Integrating a new component, however, requires options not normally available on the standard Code Generation Options dialog. (Figure 6-10 shows both dialogs.) In order to display the options required, you must merge two Builder Engine resources into the current X resource database.

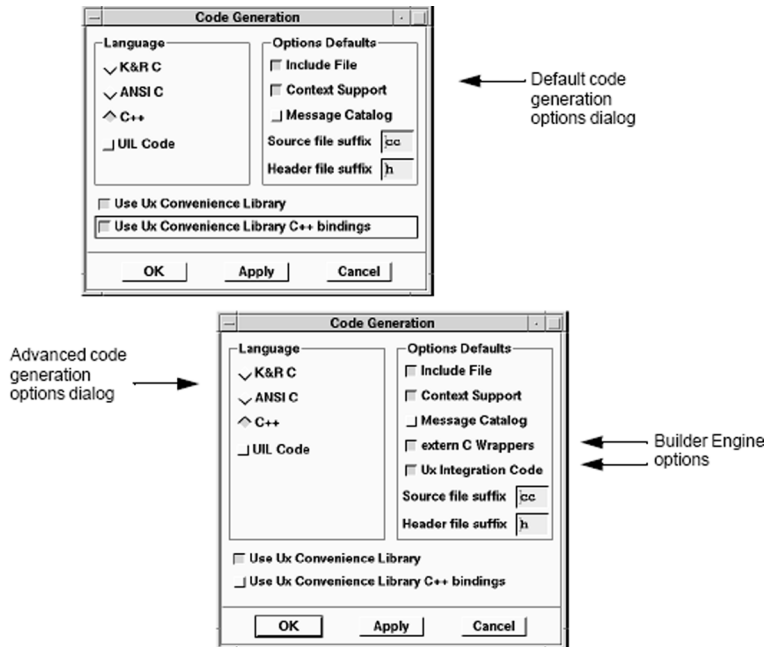


Figure 6-10 Standard and Advanced UIM/X Code Generation Options Dialogs

As part of integrating the File object into UIM/X, you will use UIM/X to generate C wrappers and Ux Integration Code. The C wrappers make a C++ class callable from a C program. The Ux Integration Code allows UIM/X to manage the component.

These advanced C++ code generation options become available in the UIM/X Code Generation Options dialog when two resources are set to true:

```
Uimx3_0*UxPrjOptionsCGenGenCWrappers.set:true
Uimx3_0*UxPrjOptionsCGenGenUxIntCode.set:true
```

It is simply a matter of merging the above resources into the current X-resource database prior to starting UIM/X. This shall be done as part of this step.

To Restart UIM/X With Builder Engine Resources

Before you begin this section of the tutorial, load the required Builder Engine resources and set up a new directory as follows:

1. Under the `chap6` directory, make a directory to store the files you will create in this section of the tutorial:

```
mkdir sect3
```

2. Change to the directory you just created:

```
cd sect3
```

3. In a terminal window, copy the File interface, `File.i`, from the `chap6/sect2` directory into your current directory:

```
cp ../sect2/File.i .
```

4. Create a directory to hold the Motif class definition you will generate, and another to hold the augmented UIM/X:

```
mkdir motif augment
```

5. Add the required Builder Engine resources to the resource database:

```
xrdb -m
```

```
Uimx3_0*UxPrjOptionsCGenGenCWrappers.set:true
```

```
Uimx3_0*UxPrjOptionsCGenGenUxIntCode.set:true
```

When you are through typing, press `Ctrl-d` to end your `xrdb` session

6. Start UIM/X from your current directory:

```
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 &
```

After a brief pause, a copyright notice window appears on the screen, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and palette appear.

7. Load the File object, `File.i`, by choosing File⇒Open in the Project Window.
8. Iconify the terminal window.

Note: To restart this tutorial, begin again from Step 2 above.

Step #10: Creating the New Class Code

Before you can integrate a new component into UIM/X, you must create a class definition for the component. The class definition must be in C++, must contain C wrapper functions, and must contain integration code. For components built in UIM/X, creating the C++ class for it is a simple matter of generating the C++ code, as you would for any project. While creating the class definition is automatic, you must be sure to select the correct code generation options.

UIM/X recognizes component class definitions provided they meet certain specifications. First, the class definition must be in C++. UIM/X can integrate C++ class definitions only, but can generate C or C++ project code using the code. Second, it must be “wrapped” in a C wrapper function. This allows the C++ class code to be linked to a C language program. More importantly, since UIM/X is a C program, it allows UIM/X itself to link with the class definition.

UIM/X also requires that the class definition contain integration code, in a specific format. This code enables UIM/X to manage the new class throughout the design process. In addition to managing the component during design-time, UIM/X must know what component-specific properties and event callbacks to display in the Property Editor, for example. This information is contained in the Ux Integration Code.

For components built using UIM/X, producing the correct class definition—in C++, and including C wrappers and Ux Integration Code—is a matter of selecting the correct code generation options and generating the “project code” based on the actual component. For a component produced outside UIM/X you must write the integration code by hand, as described in *UIM/X Advanced Topics*.

In this step you will create the File object’s C++ class definition by generating the project code using the File object just loaded. In the next step, you will integrate the new class into UIM/X.

To Create the New UIM/X Class Code

The first step in integrating a new class into UIM/X is to generate the class code, containing wrapper code, and Ux Integration code.

1. Open the Code Generation Options dialog by choosing Options⇒Code Generation in the Project Window.
2. Ensure the following radio buttons and toggle buttons are selected:
 - C++
 - extern C Wrappers
 - Ux Integration Code
3. Close the dialog by clicking OK.
4. Choose File⇒Generate Project Code on the Project Window menu. The Generate Code window appears, as shown in Figure 6-11.

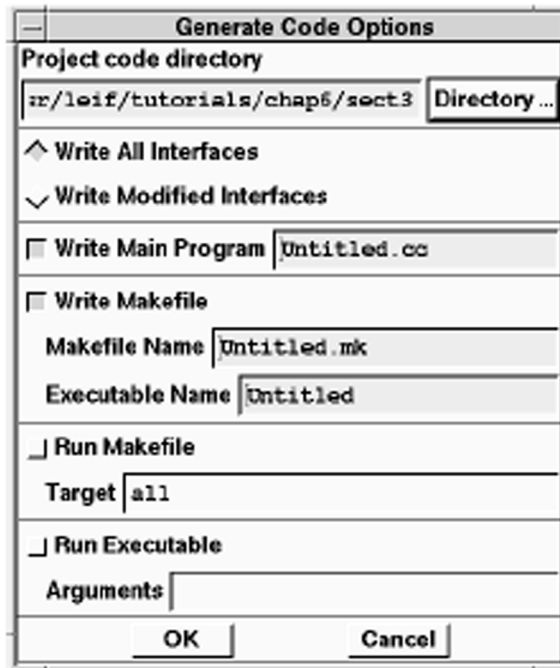


Figure 6-11 Generate Code Window

5. Check that the following radio buttons and toggle buttons are selected:
 - Write All Interfaces
 - Write Main Program
 - Write Makefile
6. Change the makefile name from its default to `File.mk`.
7. Click on OK.
UIM/X writes the files, displaying progress messages in the Messages area of the Project Window.
8. When complete, exit UIM/X.

Step #11: Compiling the New UIM/X Class Code

Once you have generated the C++ class definition for the File object you can compile the code. Two compiled versions of the class definition are required—one for linking with UIM/X, and one for linking with other programs (such as those you create *using* UIM/X). To compile a specific version, you edit the class definition makefile and set C++ compiler flags.

In order to link with UIM/X, you must inform the C++ compiler that the class code contains C wrapper functions, Ux Integration Code, and that it should be compiled with design-time management considerations.

To link with other applications you need to compile the code with the C wrapper functions only. This will produce the object file for linking with C or C++ applications, such as the project code you normally generate when using UIM/X.

The `CPLUS_CFLAGS` macro in the makefile defines compiler flags that control the object file produced. Table 6-4 explains the flags you should use to produce the object file you want:

Table 6-4 Compiler Flags Required

To Link With	Use These Flags
UIM/X	<code>-DEXTERN_C_WRAPPERS -DUX_C -DDESIGN_TIME</code>
C++ code only	No flags required.
C or C++ code	<code>-DEXTERN C WRAPPERS</code>

The meaning of the flags is explained in Table 6-5:

Table 6-5 Meaning of Compiler Flags

Compiler Flag	Meaning
<code>-DEXTERN_C_WRAPPERS</code>	Informs the compiler that C++ code is being compiled, but that the C wrapper functions for the code should be used, so it can be called by both C and C++ applications.
<code>-DUX_C</code>	Compiles the Ux Integration Code required by UIM/X.
<code>-DDESIGN_TIME</code>	Compiles for proper design-time control and presentation of the class.

In this step you will edit the File object class makefile and compile it twice—once for use with UIM/X, and once for use with other applications. In the next step you will augment UIM/X with the class definition executable.

For Linking with C and C++ Applications

1. Using a text editor, open the File object makefile, `File.mk`.
2. Find the line that defines the C++ compiler flags used:

```
CPLUS_CFLAGS = ...
```

3. Change it to read as follows:

```
CPLUS_CFLAGS = ... -DEXTERN_C_WRAPPERS
```

The three dots indicate the presence of platform-specific information. Leave that information as is. Do not type three dots. The meaning of `-DEXTERN_C_WRAPPERS` is explained in Table 6-5.

4. Save your changes and exit the text editor.
5. Compile the object code for linking with applications by typing the following at the UNIX command line:

```
make -f File.mk File.o
```

6. Move the object code produced to the `motif` directory for later use.

```
mv File.o motif
```

7. If you plan on generating C code for your projects, compile a version of `UxInterf.o` for linking with `File.o` in C applications:

```
make -f File.mk UxInterf.o
```

8. Move the object code produced to the `motif` directory for later use.

```
mv UxInterf.o motif
```

For Linking with UIM/X

In this step you will edit the File makefile once again, to include the flags required for linking with UIM/X itself. In addition, to compile the integration code UIM/X requires header files located in the `/custom/include` directory.

1. Using a text editor, open the File object makefile, `File.mk`.
2. Find the line that defines the C compiler flags used:

```
UX_CFLAGS = ...
```

3. Change it to read as follows:

```
UX_CFLAGS = ... -I$(UX_DIR)/custom/include
```

As in the earlier instruction, the three dots indicate you should leave the existing information as is.

4. Find the `CPLUS_CFLAGS` line, this time changing it to read as follows:

```
CPLUS_CFLAGS = ... -DEXTERN_C_WRAPPERS -DUX_C
-DDESIGN_TIME
```

The meaning of `-DEXTERN_C_WRAPPERS`, `-DUX_C` and `_DDSIGN_TIME` are explained in Table 6-5.

5. Compile the object code for augmenting UIM/X by typing the following at the UNIX command line:

```
make -f File.mk File.o
```

6. Move the object code to the `augment` directory for later use.

```
mv File.o augment
```

Step #12: Augmenting UIM/X

Once you have compiled the UIM/X class code, you can augment the UIM/X executable to include the new class. Augmenting UIM/X is a simple matter of making a copy of the UIM/X main program file and its makefile, then editing the makefile to include the class object file generated earlier.

When you compile the code, the new class code will be added to the UIM/X executable. Since the object file contains the class definition, as well as integration code, the resulting executable will contain all the information it needs to manage the new class.

To Augment UIM/X

1. Change to the directory containing the object code for augmenting UIM/X:

```
cd augment
```

2. Copy the UIM/X main program file into the augment directory:

```
cp uimx_directory/config/uimx_main.cc .
```

3. Copy the UIM/X makefile into the augment directory

```
cp uimx_directory/config/Makefile.uimx .
```

4. Using a text editor, open the UIM/X makefile, `Makefile.uimx`.

5. Find the line that defines the object files included in the executable:

```
APPL_CPLUSOBS = $(AUGMAINOBJ)
```

6. Change it so the executable includes the File object code, `File.o`:

```
APPL_CPLUSOBS = $(AUGMAINOBJ) File.o
```

7. Create the augmented UIM/X by running the makefile:

```
make -f Makefile.uimx uimx_aug
```

Step #13: Creating a New UIM/X Palette

Now that you have created the UIM/X augmented executable with the new File class, the File object is available for use. To make it easier to use, you can add the object to the palette. To do so, you must run the augmented executable, and create an empty subclass to hold the File object. Once created, you populate the subclass with the object by setting its declaration properties. Next, you add the component to the palette, then save the palette for future use.

Making the File Object Subclass Visible

In order to allow users to use the File object in projects, it must be made visible. To create the File object you create an empty instance and add the File object to it.

1. Run the augmented version of UIM/X:
`uimx_aug &`
2. Create a Manager, such as Drawing Area as shown in Figure 6-12:

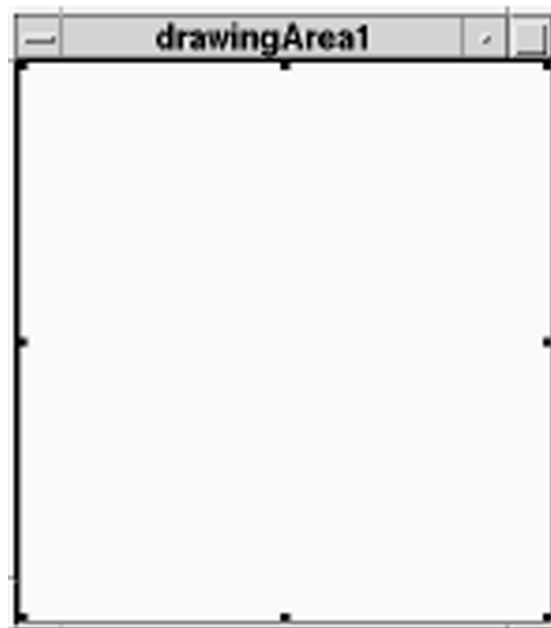


Figure 6-12 Temporary Top-Level Interface

The Drawing Area manager shall be used to temporarily hold the File subclass.

3. Create an empty instance by choosing Selected Objects⇒Instance and dragging and drawing it on the Manager, as shown in Figure 6-13:

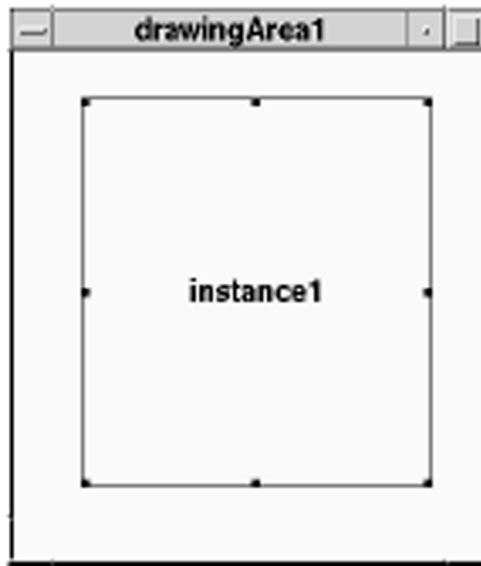


Figure 6-13 The Empty Instance, instance1

Make the instance the size you want the File object's outline to appear when it is dragged and dropped from the Palette. The size is otherwise unimportant, since a Non Visual object's dimensions are not used.

4. Open the Interpreter window by choosing Tools⇒Interpreter in the Project Window.
5. Choose Module⇒Selected Interface in the Interpreter or click on the corresponding icon .

The Interpreter title bar changes to reflect the new scope.

6. Enter the following code in the Interpreter window.

```
extern "C" swidget create_File( swidget, char * );
```

When you generated the object file for linking with UIM/X, recall that `create_File()` was defined as a C function. This was so it could be used by UIM/X to generate C or C++ code. In this step the interpreter is in C++ Mode. The above declaration is required to inform UIM/X that the create function is a C function.

7. Triple-click the line of code to highlight it, then choose Interpret⇒Declare.

The Interpreter evaluates the code, pops up the Command Line interface, and prints the following to the Interpreter Messages Area:

Result: OK

8. Close the Interpreter by choosing File⇒Close.
9. To rename the instance to something more suitable, begin by loading the empty instance into the Property Editor.
10. In the Declaration category, change the Name property to FileObj and click on Apply.

The instance should now appear as shown in Figure 6-14.



Figure 6-14 The Empty Instance, Renamed

11. Locate the properties shown in Table 6-6, and give them the values indicated.

Table 6-6 Declaration Property Values for File Object Subclass

Property Name	New Value
ArgDefinition	"swidget UxParent; char *filename;"
Component	"File"

Table 6-6 Declaration Property Values for File Object Subclass

Property Name	New Value
Constructor	"create_File"
HeaderFile	"File.h"

12. Click on Apply.

The instance inherits its properties from the named component. Since it stems from a Non-Visual object, the instance disappears.

13. Close the Property Editor by choosing File⇒Close.

14. Open the Browser by choosing Selected Objects⇒Tools⇒Browser.

Although the Non-Visual object isn't visible in the interface, it is visible in the Browser. You will use the Browser in the next step.

15. Save your work.

Adding the New Subclass to the Palette

Adding the new subclass to the Palette is a simple matter of placing the Palette in edit mode, creating a new category of components, and dragging and dropping an instance of the subclass into a category. For the sake of organization, it is helpful to create a new category to store the widgets or objects you add.

1. Make sure you are in Edit mode by choosing Mode⇒Edit from the Palette.

2. Create a new category of widget by choosing Edit⇒Create Category from the Palette.

In the dialog that appears, name the category "Other" and click on OK.

3. Display the new category by scrolling the Palette to the bottom.

4. Put an instance of the File interface into the Palette by dragging and dropping it from the Browser.

Use the Adjust mouse button, as if you were moving the interface.

5. Switch back to create mode by choosing Mode⇒Create from the Palette.

6. Save the new Palette by choosing File⇒Save As from the Palette.

Name the new Palette `File.pal`, and save it in the `thechap6/sect3/augment` directory.

7. Your new Palette should now appear as shown in Figure 6-15 (some categories have been collapsed for display purposes):

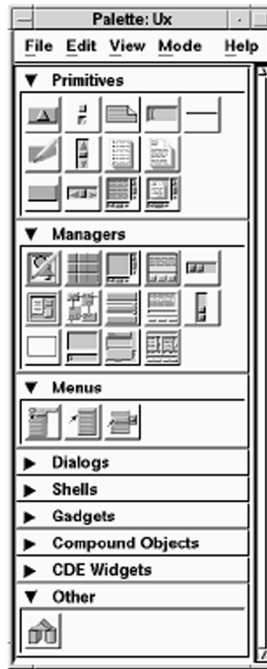


Figure 6-15 The Palette Containing the New Category and Component

8. Exit the augmented UIM/X by selecting File⇒Exit from the Project Window. You will be asked to confirm your exit. Exit without saving.

Step #14: Polishing the Augmented UIM/X

Once UIM/X has been augmented with the new class definition and an instance of the object has been added to the palette, the File object is ready for use. Further refinements are possible. By adding a few lines of code to the UIM/X main program source code, you will accelerate display of the new object and ensure the interpreter recognizes its methods. Also, you can have the palette containing the File object loaded automatically at start-up. These simple changes are made by editing the UIM/X main program file, and the UIM/X resource file.

To accelerate display of the new File object, you must preregister its class with the interpreter. This is a matter of adding code to the UIM/X main program

file. First, you add a declaration for the create function of the object's constructor. This is a function of the form `create_object()`. Then you add a call to `UxRegisterFunction`, to preregister the create function.

To enable the interpreter to recognize the object's methods you must load the File header file at start-up. To do so, you edit the main program file, adding a call to `UxLoadGlobalInclude`, the function that loads header files into the interpreter.

To load the new Palette at start-up you must add information to the UIM/X resource file, `uimx_directory/app-defaults/Uimx3_0`. The resource file must also contain settings to ensure the interpreter knows the C++ compiler flags used when the class code was generated:
`-DEXTERN_C_WRAPPERS` and `-DUX_C`.

In this step you will edit the UIM/X main program file to preregister the File create function, `create_File()`, for quick access by the interpreter. Before recompiling UIM/X, you will also rename the executable to avoid overwriting the augmented executable already created. This step also includes modifying the UIM/X resource file to load the new palette at start-up.

Editing the UIM/X Main Program File

In this step you add the preregistration code to the UIM/X main program file.

- Using a text editor, open the UIM/X main program file, `uimx_main.cc` (in the `augment` directory).
- Find the declaration for `UxRegisterFunction()`:

```
void UxRegisterFunction UXPROTO(( char *, void*
    ));
```
- Just after it, add a declaration for the File create function, `create_File`:

```
swidget create_File UXPROTO(( swidget, char *));
```
- Now locate the body of `UxRegisterFunctions()`, and add the following code, after the last call to `UxRegisterFunction()`:

```
UxRegisterFunction( "create_File", create_File);
```
- Find the section containing initialization code:

```
/* Insert initialization code for your application
    here *
```
- Add a call to load the File object's header file:

```
UxLoadGlobalInclude( "File.h" );
```
- Save your changes and exit the editor.

Editing the Makefile and Building the New Executable

In this step you edit the makefile to rename the executable to `newuimx`. Then you will run the makefile to compile the polished executable that preregisters the File class.

- Using a text editor, edit the makefile, `Makefile.uimx`.
- Locate the line that defines the executable name:

```
AUGEXEC = uimx_aug
```
- Rename it to `newuimx`, as follows:

```
AUGEXEC = newuimx
```

Renaming the executable is a safety precaution. If you have to go back a few steps, the augmented UIM/X created earlier, `uimx_aug`, will still be intact.
- Save the makefile and exit the editor.

5. Build the augmented executable by running the makefile:

```
make -f Makefile.uimx newuimx
```

Loading the Palette at Start-up

In this step you edit the UIM/X resource file to ensure the interpreter knows what compiler flags the class code was compiled with, and which palette to load at start-up.

1. Copy the UIM/X resource file into the current directory (augment):

```
cp uimx_directory/app-defaults/Uimx3_0 .
```

2. Change the permissions to make it writable:

```
chmod a+w *
```

3. Using a text editor, open the resource file, Uimx3_0.

4. Locate the lines that determine the palette loaded at start-up:

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

5. Change start-up palette resources to the new palette in the current directory:

```
Uimx3_0*UxPalettePath.value: current_directory
Uimx3_0*UxStartingPalettes.value: File.pal
```

Be sure to specify the path from the root directory, starting with a forward slash (“/”).

6. Locate the line that informs the interpreter of the CFLAGS used to compile the code:

```
Uimx3_0.cflags: ...
```

The ellipsis [...] indicates platform-specific information. Do not type three dots or delete the information.

7. Add `-DEXTERN_C_WRAPPERS-DUX_C` to the CFLAGS macro:

```
Uimx3_0.cflags: -DEXTERN_C_WRAPPERS-DUX_C ...
```

8. Save and close the updated resource file.

Section IV: Using the Integrated File Object

In this section you will test the new executable to see that the File object you added behaves as expected. As part of the test, you will use the To Do List project, this time adding an instance of the new File object to it.

This section illustrates the advantages of augmenting the UIM/X executable to include new objects. For example, adding a File object to the interface can be done by dragging and dropping from the palette. While this is the case with objects provided as a start-up project (or loaded as an interface) there is one important difference. Since the File object in the palette is already an instance, it's methods are available for use, but cannot be modified.

This section takes about 30 minutes to complete. It contains the following steps:

Step #15: Starting the New Augmented UIM/X

Step #16: Adding a File Object to the To Do List Project

Step #17: Modifying the To Do List Menus

Step #18: Testing the Integrated Project

Step #19: Generating the Code and Running the Executable

Where You Are in the Tutorial

Section I: Creating a Non-Visual File Object

Section II: Using the File Object in the To Do List

Section III: Integrating the File Object into UIM/X

⇒Section IV: Using the Integrated File Object

Step #15: Starting the New Augmented UIM/X

In this step you will start the new version of UIM/X to see that the palette you created is loaded at start up. You will also load the start-up project.

1. Under the `chap6` directory, make a directory to store the files you will create in this section of the tutorial:

```
mkdir sect4
```

2. Change to the directory you just created:

```
cd sect4
```

3. Copy the File object's header file from `chap6/sect3` into your work directory:

```
cp ../sect3/File.h .
```

4. Copy the To Do List project files into your work directory:

```
cp uimx_directory/contrib/ToDoList/* .
```

5. Change the permissions on the project files you copied to make them writable:

```
chmod a+w *
```

6. At the UNIX prompt, set the XAPPLRESDIR environment variable to the directory containing UIM/X's resource file (the one you copied and added to):

```
setenv XAPPLRESDIR ../sect3/augment
```

By default, UIM/X searches for application defaults in a number of directories in a specific order. Setting XAPPLRESDIR ensures the resource file you created, `Uimx3_0`, is the one used.

7. Run the augmented UIM/X executable by typing the following in the terminal window:

```
../sect3/augment/newuimx &
```

After a brief pause, a copyright notice window appears on the screen, to show that UIM/X is being initialized. When UIM/X is ready, the Project Window and your palette appear. Notice the palette contains the new Other category, and the object you created.

8. Iconify the terminal window.

Step #16: Adding a File Object to the To Do List Project

In this step you will load the To Do List project, and add a `FileObj` from the new palette.

1. Choose `File⇒Open` in the Project Window and choose `ToDo.prj` (in `chap6/sect4`).

2. Select OK.

Dialogs appear indicating that you are loading an interface originally created in Novice Mode, and that compound objects will be converted to widgets.

3. Dismiss each dialog as it appears by clicking Replace.

4. Add a `FileObj` from the Others category of the Palette to the To Do List interface.

5. Open the Browser by choosing `Selected Objects⇒Tools⇒Browser`.

6. Load the File Object, `FileObj1`, into the Property Editor by selecting it in the Browser and choosing Selected Objects⇒Tools⇒Property Editor.
7. In the Core category locate the `filename` property, changing it from `NULL` to `"todo.out"`.
This property specifies the file that will be operated upon. It is passed into the File Object via its create function.
8. In the Declaration category locate the `Name` property, changing it from `FileObj1` to `todoFile`.
9. Apply your changes by clicking on Apply in the Property Editor.
10. Close the Property Editor by choosing File⇒Close.
11. Save your work as a new project, `Integrated.prj`.

Step #17: Modifying the To Do List Menus

In this step you will change the callbacks for the File⇒Open and File⇒Close items to use the methods defined for the File object instead.

1. Select the menu bar and open the Menu Editor by choosing Tools⇒Menu Editor.

The Menu Editor appears, as shown in Figure 6-16.

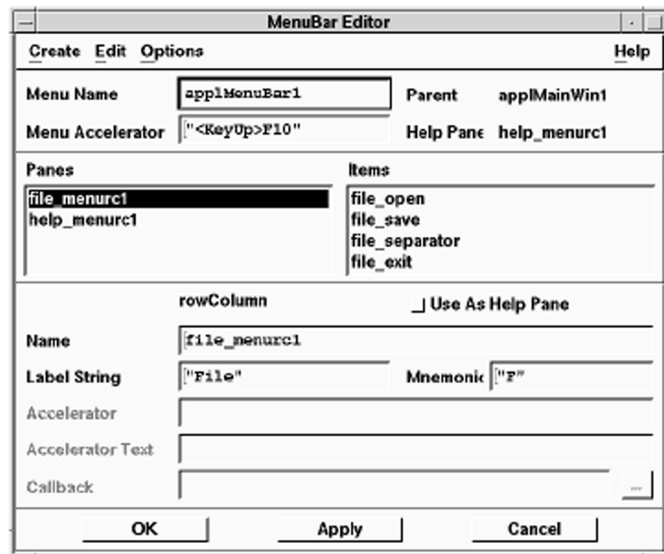


Figure 6-16 Menu Editor

2. Select the `file_open` item. The properties and callbacks are listed in the display area.
3. Click on the Text Editor button (...) beside the `Callback` property.
A Text Editor appears, displaying the code included with the To Do List project.

4. Delete the code provided, replacing it with the following code:

```
char *str;
    Widget w;
    XmString xms;

if ( File_Open(todofile, "r", &UxEnv) )
{
w = UxGetWidget(scrolledList1);
XmListDeleteAllItems(w);
while ( str = File_Readline(todofile, &UxEnv) )
{
/* remove trailing newline and add to the list */
str[strlen(str)-1] = '\0';
xms = XmStringCreateSimple(str);
XmListAddItem(w, xms, 0);
XmStringFree(xms);
}
File_Close(todofile, &UxEnv);
}
```

5. Close the Text Editor and copy the code to the callback by clicking on OK.
6. Apply the changes without closing the Menu Editor by clicking on Apply.
7. Similarly, enter the following callback for the `file_save` item.

```
char *taskList;
char *processList;

if ( File_Open(todofile, "w", &UxEnv) )
{
taskList = UxGetItems(scrolledList1);
if (taskList)
{
/* Replace commas by newlines and write out to the
list */
processList = taskList;
while (*processList)
{
if (*processList == ',')
{*
processList = '\n';
}
processList++;
}
File_Writeline(todofile, taskList, &UxEnv);
File_Close(todofile, &UxEnv);
}
}
```

8. Apply the changes and close the Menu Editor by clicking on OK.
9. Save your work.

Step #18: Testing the Integrated Project

Before generating the project code for the new To Do List project, test that the interface operates as expected.

1. As before, switch to Test Mode by clicking on the Test icon in the Project Window.

The Palette and any other open editors disappear. The Project Window and your interface remain.

2. Test the To Do List interface:
 - To add a task to the work area, type your text in the Task field, and click on Add.
 - Use the slider to assign the task a priority level.
 - To edit an existing task, highlight it in the work area, and click on Edit. The task is copied to the Task area.
3. Test the File functionality:
 - Choosing File⇒Save writes the contents of your to do list to the file `todo.out`.
 - Choosing File⇒Open displays the contents of `todo.out` in the work area.
4. You can verify that a file has been created by looking in your working directory from the command line.
5. When you are through, switch back to Design Mode by clicking on the Design icon

Step #19: Generating the Code and Running the Executable

The final step in creating your project is to edit its makefile template and generate code for the integrated To Do List project.

Editing the Makefile Template

When generating code, UIM/X uses a makefile template, replacing variables in the template with the names of elements in your project. In this step you will edit the makefile template, adding the path to `File.o` (Motif version) you created earlier. You must also inform the compiler that the object file was created with C wrappers. The instructions vary slightly, depending on whether you will be generating K&R C, ANSI C, or C++.

1. Open the Program Layout editor by choosing Tools⇒Program Layout in the Project Window.
2. Click on the Text Editor button [...] next to the Ux Makefile field. The Text Editor appears.
3. Locate the line that begins APPL_OBJS and, placing the cursor at the end, add the following:

```
APPL_OBJS = ... ../sect3/motif/File.o
```

For clarity, the part you type is indicated in bold. The three dots indicate you should leave the rest of the text as is. Do not type the three dots.

4. If you will be generating C code, add the path to UxInterf.o (that you created earlier) as well:

```
APPL_OBJS = ... ../sect3/motif/File.o
             ../sect3/motif/UxInterf.o
```

5. If you will be generating C++, locate the line that begins CPLUS_CFLAGS. Placing the cursor at the end, add the following:

```
CPLUS_CFLAGS = ... -DEXTERN_C_WRAPPERS
```

Once again, the part you type is indicated in bold. Leave the rest as is.

If you will be generating K&R C, add the above information to the end of the line that begins KR_CFLAGS instead. For ANSI C, add it to the line that begins ANSI_C.

6. There is one last change for those who will be generating C code. Since the Non-Visual Object you created is a C++ object, you must use the C++ linker. Locate the lines that define the linker used:

```
LD = \
@`if [ "$(LANGUAGE)" = "C++" ]; then echo
    $(CPLUS_CC); fi` \
`if [ "$(LANGUAGE)" = "ANSI C" ]; then echo
    $(ANSI_CC); fi` \
`if [ "$(LANGUAGE)" = "KR-C" ]; then echo
    $(KR_CC); fi`
```

- Change it to read as follows:

```
LD = \
@`if [ "$(LANGUAGE)" = "C++" ]; then echo
    $(CPLUS_CC); fi ` \
`if [ "$(LANGUAGE)" = "ANSI C" ]; then echo
    $(CPLUS_CC); fi ` \
`if [ "$(LANGUAGE)" = "KR-C" ]; then echo
    $(CPLUS_CC); fi `
```

The changes are indicated in bold.

- Close the Text Editor by clicking OK.
- Save your changes and close the Program Layout Editor by clicking OK.
- Save your work.

Generating the Code and Running the Executable

- Check that you are in Design Mode.
- Choose Options⇒Code Generation on the Project Window menu. The Code Generation Options window appears, as shown in Figure 6-17.

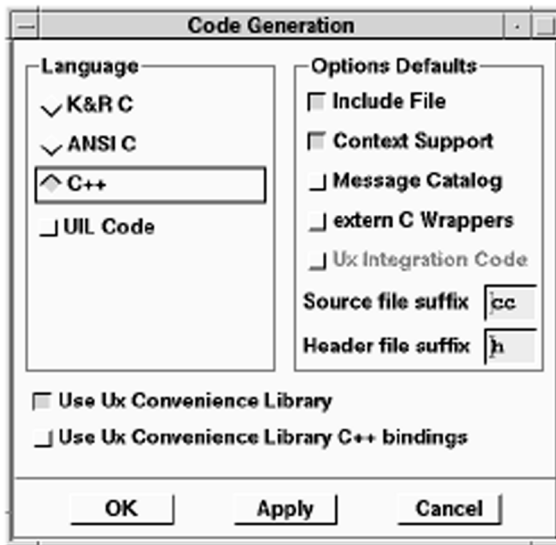


Figure 6-17 Code Generation Options

3. Ensure the following radio and options buttons are selected:

- C++
- Include File
- Context Support
- Use Ux Convenience Library

Since multiple copies of the dialogs can be created by the application, context support is required. If you carried out the steps required for generating K&R C or ANSIC, you can select those languages too.

4. Save your changes and close the dialog by clicking on OK.

5. Choose File⇒Generate Project Code As on the Project Window menu.

6. Check that the following radio buttons and toggle buttons are selected:

- Write All Interfaces
- Write Main Program
- Write Makefile
- Run Makefile
- Run Executable

7. Click OK to generate your code.

UIM/X writes the files, runs the makefile, compiles the generated code, and runs the executable. Progress messages are displayed in the Messages area of the Project Window.

8. Test your program. Verify that it works as it did in Test Mode.

9. To stop the program choose File⇒Exit.

10. Save the changes to your program.

Index

Symbols

.i file 17, 49
.prj file 17, 49

A

Adjust mouse button xii
Alt key xi
APPL_CPLUSOBS 222
application defaults xiii
Application Window 44
application window behavior 105
arranging widgets 19

B

be.rf
See also Builder Engine resource file
behavior
 adding callback 146
 application window 105
 See also Connection Editor
Browser 63
Builder Engine resource file 215
Bulletin Board widget 136

C

C Mode 169
C++ bindings 169
C++ compiler flags 219, 220
C++ Mode 169
callback accessor 37, 173
cancelling operations 83
category option menu 142
circle.xpm 109

classes

 and inheritance 154
 compiling class code 219
 compiling for C and C++ 219, 220
 compiling for UIM/X 221
 creating class definition 217–219
 creating empty 219
 creating new 154
 creating new widget class 168
 defining C++ class in code 163
 exposing behavior in class instances 155
 exposing properties in class instances 154
 See also RGB Color Editor project
clientAutoPlace resource 6
code generation 71, 72, 150, 187
 setting options 215
Color Database 90
Color Editor 89
Color Viewer 89
ColorBox project 2–35
 adding callback behavior to Push Buttons 31
 adding Push Buttons 16
 changing labels 26
 Connection Editor 31
 Constraint Editor 2, 23
 description of GUI 3
 Novice Mode 3
 Property Editor 2
 running the application 34
 Test Mode 3
 testing 34
Command Line project 133–151
 adding an option menu 139
 adding behavior to option menu 146
 adding behavior to Toggle Buttons 146
 adding Declarations 143

Index

- code generation 150
- description of GUI 134
- options menus 139, 146
- Property Editor 141
- testing 149
- Toggle Button 146
- Communication project 37–74
 - adding behavior to menus 67
 - adding callbacks to dialogs 57
 - adding dialog instances 61
 - changing dialog properties 55
 - code generation 71
 - description of GUI 38
 - menus 65
 - popping up dialogs 73
 - testing 71
- compound objects
 - definition x
- Connection Editor 31, 32, 67, 177, 184
 - and exposed callbacks 178
 - and interface methods 178
 - editing connections 184
 - loading widgets 33
 - reordering connections 181
- Constraint Editor 2, 23
- CPLUS_CFLAGS
 - See also C++ compiler flags*
- create function
 - adding parameters 193
- D**
- DDESIGN_TIME 220
- Declaration Editor 143, 165, 196
- definition
 - compound object x
 - interface xi
 - Motif widget x
 - object x
 - project xi
- DESIGN_TIME 198
- DEXTERN_C_WRAPPERS 219, 220
- dialogs
 - centering over calling interface 57
 - Message Box 76
 - popping up 38, 73
 - setting display size 64
 - unmanaging Push Buttons 58
 - See also Communication project*
 - See also Drawing Editor project*
 - dragging and drawing widgets (detailed) 42
 - dragging and dropping widgets 47, 82
 - (detailed) 10
- Drawing Area widget 9, 157
- Drawing Editor project 76–132
 - adding a pulldown menu 95
 - adding color-drawing behavior 92
 - adding dialogs 118–127
 - adding line-drawing functionality 105–118
 - application window behavior 76, 105
 - cascading menu 97
 - color-changing Push Buttons 86
 - description of interface 77
 - generating code 127–132
 - Menu Editor 76, 95
 - menus 94–105
 - Message Box widget 76
 - methods 76
 - property accessor methods 122
 - pulldown menus 95
 - testing color-changing Push Buttons 93
 - testing line-drawing Push Buttons 117
 - testing menus 104
 - Translation Table List 110
 - translation tables 76
 - UxPutTranslations() 117
- duplicating widgets 19–22, 48
- DUX_C 220
- E**
- editors
 - Browser 63
 - Color Editor 89
 - Color Viewer 89
 - Connection Editor 31, 67, 177, 184

- Constraint Editor 2, 23
- Declaration Editor 143, 165, 196
- Event Editor 110
- Icon Viewer 107
- Interpreter 149, 185
- Menu Editor 76, 94, 95, 139
- Method Editor 37, 122, 167, 198
- Option Menu 146
- Property Editor 2, 54, 141, 142, 161, 173, 195
- Translation Table Editor 106, 110
- ellipse.xpm 109
- Enter key xi
 - See also Return key xi*
- Event Editor 110
- F**
- File Selection Box widget
 - adding behavior 217
 - See also Communication project*
- files
 - .i 17, 49
 - .prj 17, 49
- Form widget 158
- Frame widget 83
- G**
- Gadgets 138
- Generate Code Options 35, 72
- generating code 71
- H**
- Horizontal Scale widget 160
- I**
- Icon Viewer 107
- Installation Directories xi
- instances 37
 - adding dialogs to an interface 61
 - adding to an interface 168
 - and Novice Mode 61
 - and Standard Mode 61
 - exposing callbacks 37, 169, 173
 - exposing properties 37, 169
 - novice mode and standard mode differences 73
 - setting display size 63
 - setting exposed properties 175
- integrating objects
 - See also Non-Visual project*
- integration code
 - See also Ux Integration Code*
- interfaces
 - creating reusable 154
 - definition xi
 - setting resizing constraints 2
- Interpreter 149, 185
- L**
- Label widget 159
- line.xpm 108, 109
- line-drawing Push Buttons 117
- M**
- Makefile.uimx 222
- Menu Editor 76, 94, 95, 139
 - Novice Mode differences 66
- Menu mouse button xii
- menus
 - adding behavior 67
 - built-in behavior 76
 - cascading 97
 - Option Menu Editor 140, 146
 - option menus 139
 - pulldown 95
 - Selected Objects 20, 48, 137
- Message Box dialog 47, 76
 - See also Communication project*
 - See also Drawing Editor project*
- Method Editor 37, 122, 167, 198
 - and Novice Mode 39
- methods 76, 173
 - and Connection Editor 177
 - callback accessor 37, 173
 - creating 198

Index

- for integrated objects 198
- property accessor 37, 118, 122, 168
- Motif widget
 - definition x
- mouse
 - adjust button xii
 - cancelling operations 83
 - menu button xii
 - responding to general mouse activity 106
 - select button xii
 - usage xii
 - See also Translation Tables*
- mouse buttons
 - naming conventions for xii
- mouse pointer
 - compass shape 13
- moving widgets 12–15, 47
- N**
- naming conventions
 - menu options xi
 - methods 198
 - Return key xi
 - shell prompts xi
 - widget labels 17
- Non-Visual project 189–240
 - adding behavior to menus 208, 233
 - adding categories to Palette 223, 226
 - adding instance to interface 203
 - adding parameter to create function 193
 - C++ compiler flags 219
 - creating File Selection Box subclass 219
 - creating methods 198
 - description of GUI 190
 - generating code 211, 237
 - integrating into UIM/X 212–230
 - testing 211, 237
- Non-Visual Shell widget 189
- Novice Mode 3
 - and instances 61
 - and Method Editor 39
 - Menu Editor differences 66
 - Palette in 7
 - starting in 5, 39
- O**
- object
 - definition x
- object class 212
- Option Menu Editor 140, 146
- Option Menus 139
- P**
- Palette 2
 - adding categories 223, 226
 - adding items 226
 - category 7
 - expand arrow 7, 8
 - Novice Mode 7
 - using 7–8
- Project Window
 - duplicating widgets in 20
 - icons in 11
- projects
 - ColorBox project 2–35
 - Command Line project 133–151
 - Communication project 37–74
 - definition xi
 - Drawing Editor project 76–132
 - Non-Visual project 189–240
 - RGB Color Editor project 154–188
 - saving (detailed) 17, 49
- Prompt widget 76
- properties
 - adding to Property Editor 169, 212
 - Behavior 3, 145
 - changing at design-time 2, 87
 - changing at runtime 76
 - changing dialog 54
 - changing for several widgets at once 87
 - exposing in instances 169
- property accessor 37
- property accessor methods 118, 122, 168
- Property Editor 2, 52, 54, 141, 142, 161, 173, 195

- loading by name 30
- loading into 52
- Push Button widget 16
 - adding callback behavior 31, 92, 146

R

- rectangle.xpm 109
- resize grid 12
- resizing interfaces 2
- resizing widgets 12–15, 46
- resources
 - setting xiii
 - setting for advanced Code Generation options 217
- Return key xi
- RGB Color Editor project 154–188
 - changing labels 161
 - code generation 187
 - Connection Editor 177
 - description of GUI 155
 - exposing behavior in class instances 173
 - exposing properties in class instances 168
 - Property Editor 161
 - setting exposed properties 175
 - testing 185
 - using exposed behavior 178
- Row Column widget 138

S

- saving, detailed instructions 17, 49
- Scrolled Window widget 81
- Select mouse button xii
- Selected Objects popup menu 20, 48, 137
- selecting widgets 51, 87
- selection handles 11
- Standard Mode
 - and instances 61
 - starting in 79, 135, 157, 192
- subclassing
 - See also classes*
 - See also RGB Color Editor project*
- subprocess control

See also Command Line project

T

- Test Mode 3, 34, 71, 149, 185
- testing
 - ColorBox project 34
 - color-changing Push Buttons 93
 - Command Line project 149
 - Communication project 71
 - Drawing Editor project 93, 104, 117
 - Non-Visual project 237
 - RGB Color Editor project 185
- Text widget 15, 137
- To Do List project
 - See also Non-Visual project*
- Toggle Button 146
- Toggle Button gadget 138
- Translation Table Editor 106, 110
- Translation Table List 110
- translation tables 76
 - and application window behavior 106
 - and mouse activity 76, 106
 - assigning to a widget 116
- Typographic Conventions xi

U

- UIM/X
 - makefile 222
 - saving your work 49
 - starting in Novice Mode 5, 39
 - starting in Standard Mode 79, 135, 157, 192
- uimx_aug 222
- Uimx3_0*UxPrjOptions 215
 - CGenCWrappers 215
 - CGenUxIntCode 215
- Ux Integration Code 217
- UxCreateSubproc() 133, 145
- UxExecSubproc() 133
- UxGetBackground() 93
- UxGetProperty() 76, 93
- UxGetTextString() 58, 59
- UxManage() 61

Index

UxPopupInterface() 186
UxPutBackground() 93
UxPutDefaultPosition() 60
UxPutLabelString() 168, 179
UxPutProperty() 76, 93
UxPutTranslations() 117
UxSetSubprocClosure() 145
UxThisWidget() 58, 59
UxVisualInterface() 37

W

widget creation

- adding resizing constraints 23
- dragging and drawing (detailed) 10, 42
- dragging and dropping 82

widget operations

- and the Browser 64
- cancelling 83
- creating custom colors 91
- duplicating in Project Window 20
- duplication 19, 48
- moving and resizing 12–15, 46
 - using window decorations 12

widget selection

- marquee selection 87
- selecting multiple widgets 87
- selection handles 11

widgets

- adding behavior 2
- adding behavior to dialogs 57
- Application Window 42, 44
- arranging 21
- built-in behavior 2
- Bulletin Board 136
- changing properties 2, 27, 141, 161
- definition x
- dialog 76
- dragging and dropping 82
- Drawing Area 9, 157
- File Selection Box 217
- Form 158
- Frame 83

Horizontal Scale 160
Label 159
Message Box dialog 47, 76

- moving 47
- naming conventions 17

Non-Visual Shell 189
Prompt 76
Push Button 17

- resizing 46

Row Column 138
Scrolled Window 81

- setting colors 89

Text 15, 137
Toggle Button gadget 138

X

XAPPLRESDIR 232
XmMessageBoxGetChild() 60
XtUnmanageChild() 60