Sun WorkShop Visual User’s Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-3574-10
May 2000, Revision A

Send comments about this document to: docfeedback@sun.com

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Please
Recycle

Adobe PostScript™

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

CHAPTER 1

Overview

Introduction to Sun WorkShop Visual

Sun WorkShop™ Visual is an interactive tool for building graphical user interfaces (GUIs) using the widgets of the standard OSF/Motif toolkit as building blocks. Sun WorkShop Visual lets you design a hierarchy of widgets on the screen quickly and easily by clicking on icons. It displays your design in two ways simultaneously: as a tree structure which represents the widget hierarchy and as a dynamic display—an active prototype which shows what your interface looks like and how it behaves. Interactive editing features let you browse through the hierarchy, cut and paste individual widgets and change widget resources. Because the dynamic display changes as you edit your widget hierarchy, you can immediately see the effects of your actions.

When your design is complete, Sun WorkShop Visual automatically generates the code files required for your interface. You can compile, link and run the code generated by Sun WorkShop Visual without modification as a prototype of your interface. You connect the prototype interface to your application code by writing connecting code. Sun WorkShop Visual provides sample files which you can use as templates for the connecting code.

One way you can connect your Sun WorkShop Visual interface to the application is by associating callback functions with specific widgets. For example, you can designate a certain routine to be called whenever the user clicks a certain pushbutton in the interface. Callbacks let your application receive and handle user events from the interface.

Sun WorkShop Visual incorporates the tools Sun WorkShop Visual Replay, Sun WorkShop Visual Capture and AppGuru. All of these tools help you with your design. AppGuru provides you with the basics for your application while Sun WorkShop Visual Replay and Sun WorkShop Visual Capture let you update and examine your existing Motif applications. See Chapter 13 starting on page 417 for details on both Sun WorkShop Visual Capture and AppGuru and Chapter 14 starting on page 433 for details on Sun WorkShop Visual Replay.

With Smart Code callback technology, Sun WorkShop Visual helps you to build a thin client and server application from your design. It also allows you to create client applications capable of accessing the Internet. The following chapters cover this area:

- Chapter 15 starting on page 477. This describes the building blocks of Smart Code.
- Chapter 16 starting on page 485. This introduces the basic principles of Smart Code.
- Chapter 17 starting on page 501. This explains how to create a thin client and a server from your design - a tutorial is included.
- Chapter 18 starting on page 533. This chapter describes how Sun WorkShop Visual automatically generates the code and structure required for your design to access the Internet.

Sun WorkShop Visual can also be extended to use widgets from other X toolkits. Chapter 23 starting on page 627 discusses how to extend Sun WorkShop Visual to include additional widgets.

Sun WorkShop Visual typically generates code for use with the standard OSF/Motif and X toolkits. However, it can also generate code that will result in an equivalent interface on Microsoft Windows. The code can be structured in such a way that the majority of your application will remain the same on either platform. This technique makes extensive use of Sun WorkShop Visual's C++ code generation facilities and is discussed in Chapter 11 starting on page 357 and Chapter 12 starting on page 391.

Sun WorkShop Visual can also generate Java from your design, giving you an alternative cross-platform strategy. This is covered in Chapter 10 starting on page 313.

Operating Environment

Sun WorkShop Visual 6 requires one of the following configurations:

- Solaris^{TM} 2.6 Operating Environment and the Motif from the Solaris 2.6 release
- Solaris 7 Operating Environment and the Motif from the Solaris 7 release
- Solaris 8 Operating Environment and the Motif from the Solaris 8 release

Note – Solaris 2.6, Solaris 7 or Solaris 8 Operating Environment for Developer System Support or Entire Distribution cluster has to be installed.

Requirements for Compiling Generated Code

In order to compile the code generated by Sun WorkShop Visual, you will need the following:

- X11R5 or X11R6 headers and libraries
- Solaris Motif or Solaris CDE packages available with Solaris 2.6, Solaris 7 or Solaris 8

To compile the MFC code generated when in Microsoft Windows mode, you will also require the following;

- Microsoft Windows 3.1, 3.11, 95 or 98 or Microsoft Windows NT 3.5.1 or 4.0 or Microsoft Windows 2000
- Visual C++
- MFC

Basic Concepts and Terms

The following introduces some of the basic concepts of Sun WorkShop Visual together with some of the terms used in this manual.

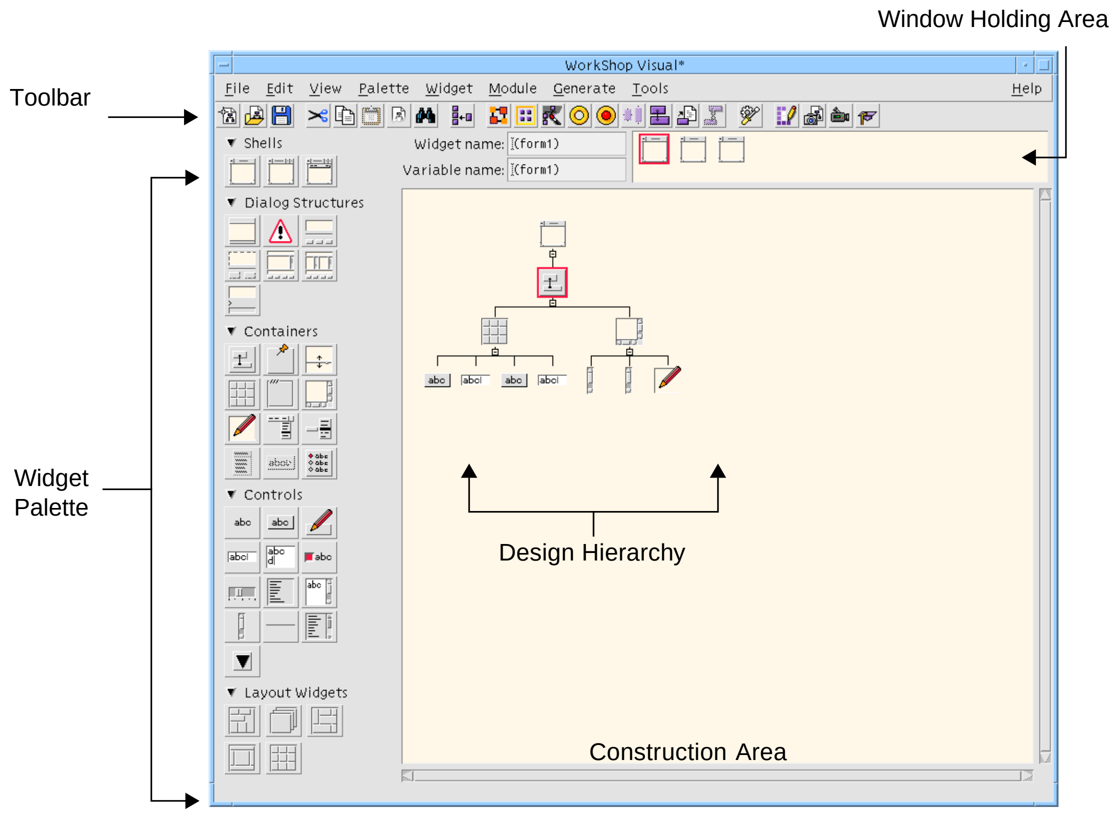
Widgets

Widgets are the building blocks used to create a user interface. Some widgets have a specific appearance and behavior in the interface display. Examples in Motif include PushButton, Label, and TextField widgets. Another type of widget is invisible itself but serves to contain and organize other widgets and is thus known as a *container widget*. Container widgets include the Form, BulletinBoard, MenuBar and RowColumn widgets.

All the Motif widgets, and any additional widgets which your configuration of Sun WorkShop Visual uses, are represented by icons in a *widget palette* on the left side of the main Sun WorkShop Visual screen. When you click on one of these icons, a widget of that type is added to the design. Individual widgets in the design are known as *instances* of a *widget class*. For example, if you click on the PushButton icon three times, you add three instances of the widget class PushButton to your design.

Design Hierarchy

The widgets in a design are organized in a *design hierarchy* which Sun WorkShop Visual displays as a tree which has its root at the top and branches spreading downward. The design hierarchy is displayed in the large drawing area of the main Sun WorkShop Visual screen, as shown in Figure 1-1. This area is called the *construction area.*



Widget name: (form1)

Variable name: (form1)

FIGURE 1-1 The Main Sun WorkShop Visual Screen

Widgets added to the hierarchy are *children* of the *parent* widget immediately above them. This relationship is important because a parent *widget* can affect its children's appearance and behavior. For example, a RowColumn widget can impose a strict layout on its children which causes the children's size and position to change automatically.

Parent widgets appear above their children on the screen, as shown in Figure 1-2.

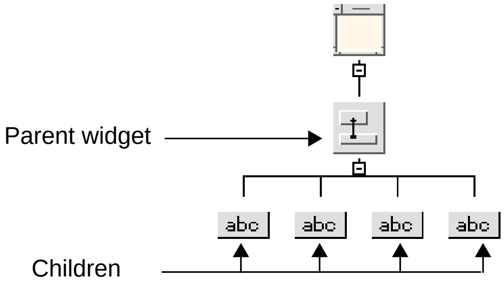


FIGURE 1-2 A Design Hierarchy

Resources

Resources are attributes of a widget which affect its appearance or behavior. Examples include dimensions, the content of a label or text field, and color. When Sun WorkShop Visual creates an instance of a widget, it assigns valid default values for each resource. Interactive *resource panels* allow you to supply your own resource values.

Since resources and their valid values are defined by the widget manufacturer, and not by Sun WorkShop Visual, it is not possible to document all of them fully in the Sun WorkShop Visual manual. However, to aid you in learning Sun WorkShop Visual, we discuss some commonly used Motif resources in this book. As you become more experienced, you should consult the Motif documentation for complete information about its widget set and the possible resource settings for each widget. If you are using widgets from other toolkits, consult the documentation from the widget developer for guidelines.

One group of resources (*attachments*) controls the spatial position of widgets within the Form container widget. Sun WorkShop Visual provides an interactive editor, the *Layout Editor*, for setting attachments.

Gadgets

Some classes of widgets have counterparts called *gadgets*. Gadgets are like widgets but have a more restricted set of resources. Because gadgets are less expensive than widgets in terms of machine resources, they are sometimes preferred.

Protection from Invalid Actions

Sun WorkShop Visual has many features designed to protect you from specifying widget hierarchies or resource combinations which are not valid in Motif. Commands that cannot be executed in the current circumstances, resources that do not apply to a particular widget and the icons of widgets that are not valid children of a proposed parent widget are *grayed out* on the screen, as shown in Figure 1-3. Grayed-out commands, resources, and icons have no effect if selected.

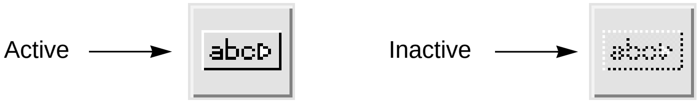


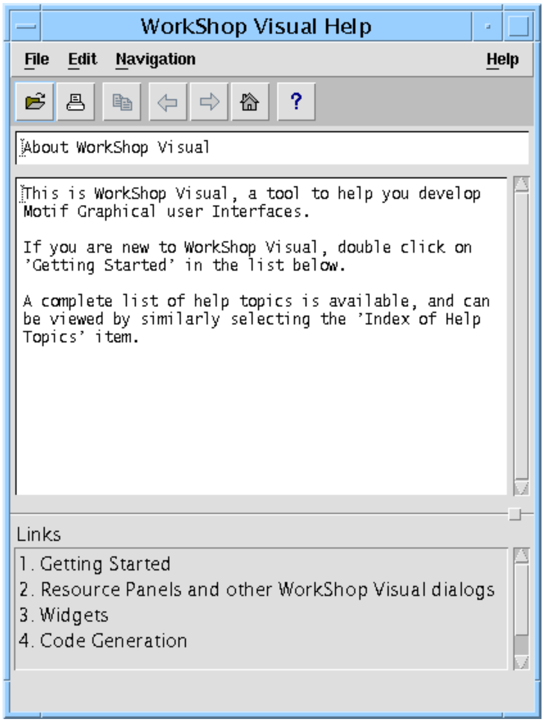
FIGURE 1-3 Active and Inactive CascadeButton Icons

Sun WorkShop Visual also rejects invalid resource settings. An entry on a resource panel may be rejected for two reasons: either the value entered is outside the valid range, or you are trying to change a resource which is controlled or limited by the widget's parent. This subject is discussed in the *Using the Resource Panels* chapter. Motif's rules for resource settings are complex and invalid settings can have serious consequences. This feature of Sun WorkShop Visual ensures that your resource settings are valid.

On-Line Help

On-line help is available throughout Sun WorkShop Visual. For general help, pull down the Help Menu in the main window and select the "Help" option. There are two viewers which can be used to display help: Sun WorkShop Visual Help and Netscape^{TM}. Use the "Viewer" pullright menu in the "Help" menu to change between them. By default, Netscape is used.

The Sun WorkShop Visual Help window is shown in Figure 1-4.



About WorkShop Visual

This is WorkShop Visual, a tool to help you develop
Motif Graphical user Interfaces.

If you are new to WorkShop Visual, double click on
'Getting Started' in the list below.

A complete list of help topics is available, and can
be viewed by similarly selecting the 'Index of Help
Topics' item.

Links

1. Getting Started
2. Resource Panels and other WorkShop Visual dialogs
3. Widgets
4. Code Generation

FIGURE 1-4 Help Viewer

Sun WorkShop Visual Help has a menubar and a toolbar. There is a “File” menu for the usual file operations, an “Edit” menu for text editing functions and a “Navigation” menu for navigation commands. The toolbar buttons are shortcuts to many of the menu items. The status line at the bottom of the window informs you of each button’s function as you move the mouse pointer over it. A list of related topics is displayed in the section at the bottom of the help viewer window. Double-click on one of these to view help on that topic.

Most dialog boxes and resource panels also have a “Help” button which you can click on for specific help about that dialog box.

The help viewer provided with Sun WorkShop Visual can be built into your application to provide help for your users. This is described in “Setting up Help in Sun WorkShop Visual” on page 605.

Using Netscape for Help

By default, Sun WorkShop Visual uses Netscape to display hypertext help. In Netscape, you can use the usual navigation keys and commands. You can also click on any highlighted word or phrase in the text to follow a link.

Widget Palette Help

The “Palette Icons” option in the main Sun WorkShop Visual Help Menu displays a copy of the widget palette with the name of each Motif widget icon. Clicking on any of the icons on this screen displays a description of the widget class of which the selected widget is a member. A list of the icons and their names is also available on the Sun WorkShop Visual Quick Reference Card.

The Sun WorkShop Visual Development Cycle

The process of creating Sun WorkShop Visual applications usually involves the following four stages:

Designing the interface. This stage includes the following operations:

- Building the widget hierarchy
- Setting resources
- Using the Layout Editor to adjust the layout
- Designating callbacks to be associated with individual widgets

Generating code. Sun WorkShop Visual automatically generates all the C or C++ code needed to display and operate your interface. Sun WorkShop Visual can also generate a *stubs file* containing all the *#include* statements and function declarations necessary to connect the interface code to your application code.

Writing code. To connect your interface prototype to a real application, supply the necessary code between the empty function brackets in the stubs file.

Linking, running and testing. This phase follows the debugging cycle needed for developing any software program.

How This Manual Is Organized

This manual is organized in three main parts:

- Tutorial
- Power use
- Reference

The Tutorial

Because Sun WorkShop Visual is highly interactive, it is easier to learn its features by actually going through the steps for a simple layout than by reading descriptions of the various features. If you are new to Sun WorkShop Visual, we recommend reading the tutorial chapters at your workstation and performing the steps given to build a simple interface. The tutorial begins with Chapter 2 starting on page 13 and continues through the following chapters, giving detailed instructions for all stages of the development cycle: building the design hierarchy, setting resources, adjusting the layout, setting callbacks, generating code and writing a very simple callback routine. At the end of the tutorial you will have a fully operational (if rudimentary) interface and you will have used all the major features of Sun WorkShop Visual.

Knowledge of the X Window System and Motif is valuable at all stages of learning Sun WorkShop Visual. However, if you are new to X or Motif, you can profit from the first several chapters of the Sun WorkShop Visual tutorial while studying the documentation for X and Motif simultaneously. The bibliography in this manual provides a list of recommended books on X and Motif. At the code generation stage you must have some understanding of X and Motif, and one of the programming languages used by Sun WorkShop Visual (C, C++, or UIL).

Power Use

The tutorial ends in Chapter 7 starting on page 207. The remainder of the manual covers advanced techniques for getting the most from Sun WorkShop Visual. It is intended for users who are familiar with Motif and X and who have prior experience with Sun WorkShop Visual or have finished working through the tutorial. There are sections dealing with the advanced code generation capabilities, Makefile generation, configuring the widget palette and toolbar, integrating user-defined widgets, advanced layout and internationalization. There are also short tutorials illustrating the structured code generation and cross platform development capabilities using MFC (Microsoft Foundation Classes) or Java. In addition, there are chapters describing the design tools Sun WorkShop Visual Replay, Sun WorkShop

Visual Capture and AppGuru together with details on how Sun WorkShop Visual Replay can be extended. There are also chapters explaining Sun WorkShop Visual's Smart Code, thin client and Internet capabilities.

Reference

The reference section is intended for Sun WorkShop Visual users at all levels. It includes:

- Summaries of all the Sun WorkShop Visual commands
- A summary of the Motif widgets and available resources, and some information on how they are mapped to Microsoft Windows code
- Suggestions for troubleshooting
- A description of the resources which can be set to alter Sun WorkShop Visual's behavior and appearance
- A description of the keywords used in Sun WorkShop Visual Replay scripts
- Some details on the integration of Sun WorkShop Visual with third party products
- A glossary

Conventions Used in this Manual

1. New terms occur in *italics* the first time they appear. These terms are defined in the Glossary.

2. The names of keyboard keys and mouse buttons appear in italics, enclosed by angle brackets: <*Tab*>. When two keys must be pressed simultaneously, we use this form:

   <1st key-2nd key>

   For example: *<Ctrl-C>, <Meta-H>, <Shift-button1>*

3. Text to be typed at the keyboard is shown in this format:

   `type this exactly`

   without quotation marks. If quotation marks appear, they are to be entered with the text.

4. Some menu commands have keyboard accelerators—keystrokes which can be used to execute the command without using the mouse. In these instructions, we mention keyboard accelerators in parentheses after naming the menu command. The following instruction:

   Pull down the Widget Menu and select “Reset” (<*Ctrl-T*>).

   means to select “Reset” from the menu or press <*Ctrl-T*>—but not both.

5. File names, function names, and variable names are all given in italics. Function names are distinguished by empty parentheses after the function name: *XtAppMainLoop()*. Angle brackets indicate a variable portion of a name:

   The default widget name is in the form <*widget-class*><*n*>, where widget-class is a Motif class (button, label etc.) and *n* is a numeral.

6. “Click” always means to use mouse button 1 unless otherwise noted. Unless you have reconfigured your mouse, button 1 is usually the left button, button 2 the middle button and button 3 the right button.

   The instruction to “click twice” is different from “double-click”. “Double-click” means that you must press the mouse button twice in rapid succession. “Click twice” can be done at any speed.

7. Names of Motif widget classes are capitalized: Label, PushButton. When similar words are used in an ordinary sense, they are not capitalized:

   The main window of this design does not use a MainWindow widget but a Form widget.

8. Books mentioned in the text of this manual are listed in Appendix E, “Further Reading”, starting on page 885.

CHAPTER 2

Building the Widget Hierarchy

Introduction to the Tutorial

The tutorial section of this document comprises the following chapters:

- Chapter 2 ”Building the Widget Hierarchy“.
- Chapter 3 ”Resources“.
- Chapter 4 ”The Layout Editor“.
- Chapter 5 ”Other Editors“.
- Chapter 6 ”Activating the Interface: Adding Your Own Code“.
- Chapter 7 ”Generating Code“.

The tutorial is meant to be read at your workstation while you follow the step-by-step instructions to build a simple interface. In the process, you will be introduced to all of the major features of Sun WorkShop Visual.

When completed, the tutorial interface looks like Figure 2-1:

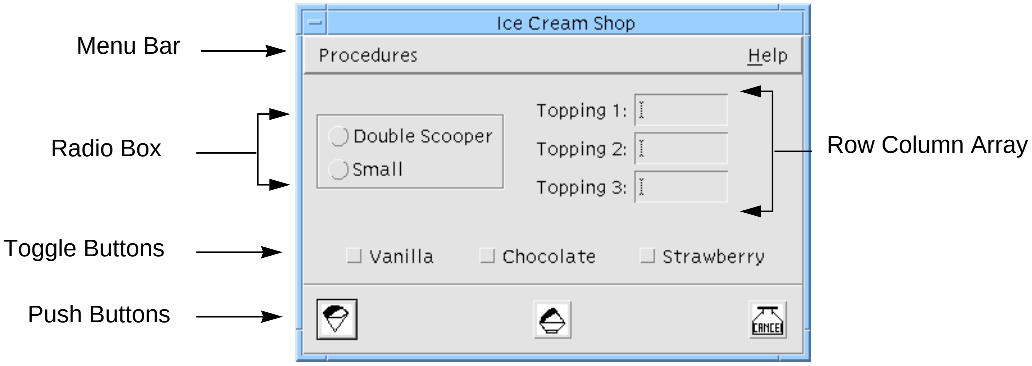


FIGURE 2-1 Tutorial Interface

Most of this tutorial addresses the Motif-only style of development, except where reference to a more general technique makes the inclusion of cross platform information relevant. There are further specialized tutorials on structured code generation and cross platform development in Chapter 9, “C++ Code Tutorial”, starting on page 277 and Chapter 12, “Creating a Microsoft Windows and Motif Application”, starting on page 391.

The Design Hierarchy

The first steps in building an interface are deciding which widgets should be used to build it and then developing an appropriate design hierarchy on the Sun WorkShop Visual screen. These steps are explained in this chapter. In the process, you will learn how to:

- Start the program and begin a new design
- Build a design hierarchy for the five common combinations of widgets shown in Figure 2-1
  - Menu bar
  - Radio box
  - Row column array
  - Group of toggle buttons
  - Group of pushbuttons
- Assign names to widgets
- Save and retrieve Sun WorkShop Visual files
- Edit the structure of a design hierarchy using cut, paste, copy and on-screen dragging facilities

Starting and Stopping

You should have Sun WorkShop Visual properly installed on your system before you begin the tutorial. Consult the installation instructions or your system administrator if Sun WorkShop Visual is not yet installed, or if the commands below do not bring up the main Sun WorkShop Visual screen.

Use one of the following commands to start Sun WorkShop Visual from within X. Either command brings up the main Sun WorkShop Visual screen.

- **Type:** `visu`

or

- **Type:** `small_visu`

`(for VGA or other small screen displays)`

Note – If you invoke Sun WorkShop Visual with `small_visu`, the icons are smaller and slightly different from those shown in this document.

Command-Line Options

Sun WorkShop Visual has numerous command-line options. These are documented in Chapter 24, “Command Line Operations”, starting on page 685. You can also get a listing of them when you invoke Sun WorkShop Visual with the `-x` option. Sun WorkShop Visual also accepts all the standard X toolkit options.

To resume work on a previously saved design, you can specify the filename on the command line: `visu <filename>.` The current filename is displayed in the window border. When there are unsaved changes the filename is followed by an asterisk (*).

To start Sun WorkShop Visual in *Microsoft Windows mode*, which enables the cross platform development capabilities, use the command line switch `-windows`.

The File Menu of Sun WorkShop Visual provides commands to let you save your work, exit the program and come back to your design later. We introduce these commands here so you can use them at any time during the tutorial.

Save, Save As

You can save your design by selecting “Save as...” (<*Ctrl+A*>) from the File Menu. “Save as...” displays a file browser, described later in this chapter, which lets you specify a filename for your design.

If you have already specified a filename, you can simply select “Save” (<*Ctrl+S*>). This procedure is faster since you are not prompted for a filename. By convention, names for Sun WorkShop Visual design files have the suffix `.xd`.

Sun WorkShop Visual’s window border contains the name of the current file. If there are any unsaved changes, the filename is followed by an asterisk (*).

In the Save As dialog you are given the option of saving a file suitable for importing into Visaj™, the visual application builder for Java. This is described in detail in “Moving Sun WorkShop Visual Designs to Visaj” on page 347.

Open, New, Exit

To load a saved design file into Sun WorkShop Visual, use the “Open” command (<*Ctrl+O*>). “Open” displays a file browser, described later in this chapter, which lets you select an existing design file. “New” (<*Ctrl+N*>) clears the construction area and starts over with an empty design. “Exit” (<*Ctrl+E*>) terminates the program. All three of these commands discard any changes you have made to the design. If you have changes in your design that have not yet been saved, Sun WorkShop Visual asks if you want to save before it executes any of these commands.

Navigating in the Menus

You can select commands from the Sun WorkShop Visual menus in three ways:

- Clicking with the mouse
- Using keyboard *accelerators*
- Using keyboard *mnemonics*

Accelerators

A keyboard accelerator is a keystroke which is designated to execute a menu command. For example, you can press <*Ctrl+S*> to execute the “Save” command.

Accelerators work whenever the input focus is in any region of the Sun WorkShop Visual screen. Accelerators are printed in parentheses wherever this tutorial instructs you to execute a command. They also appear opposite the command name in the pulldown menu on the screen.

Mnemonics

The underscored characters in menu names and options are mnemonics, a way of navigating in the menus without using the mouse. To pull down a menu by using its mnemonic character, press that character while holding down the <*Meta*> key. For example, you can pull down the File Menu by pressing <*Meta-F*>. After pulling down the menu, you can select any item by pressing its mnemonic character without <*Meta*>. The complete sequence for calling "Save" using mnemonics is: <*Meta-F*>, <*s*>.

In the *Using the Resource Panels* chapter, you will learn how to set up accelerators and mnemonics on the menu bars you create in Sun WorkShop Visual.

Toolbar

The Sun WorkShop Visual interface includes a toolbar which can be configured by the user to contain buttons corresponding to menu items. The default toolbar layout is shown in Figure 2-2.

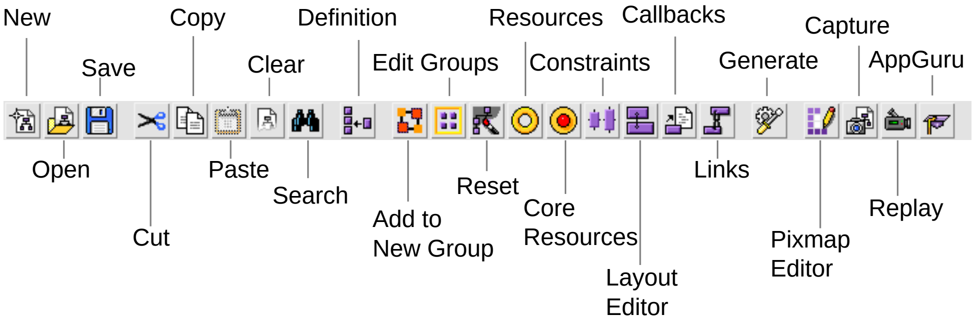


FIGURE 2-2 Default Toolbar Layout

When you select a toolbar button, it does exactly the same thing as the corresponding menu button.

Microsoft Windows mode specific elements

When Sun WorkShop Visual is in Microsoft Windows mode the default toolbar configuration contains two additional elements: the *flavor* option menu and the *Microsoft Windows compliant* toggle button. These are shown in

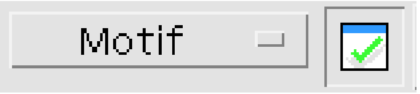


FIGURE 2-3 Windows-specific Toolbar Items

The flavor option menu is used to indicate which target code will be generated, e.g. Motif, Motif XP. This is useful when using the toolbar code generation buttons. The Microsoft Windows compliant toggle indicates whether the current design is Microsoft Windows compliant. That is, whether Sun WorkShop Visual could generate valid Microsoft Windows code for it. The toggle button is also used to invoke the compliance checking process. These features are discussed more fully in “Microsoft Windows Compliance” on page 363.

Status Line

The status line appears at the bottom of the Sun WorkShop Visual window. Prompts are displayed here when the mouse button moves over a menu item, a button in a toolbar or a button in the Widget Palette. This relates to all toolbars, including those in the Pixmap Editor and the Layout Editor. The prompts tell you what each menu item or button does. Information can only be displayed for those buttons which are enabled (or *sensitive*).

Starting the Design

All dialogs start with a Shell widget. The Shell icons, shown in Figure 2-4, are in the upper left corner of the widget palette. Although there is only one Shell widget, changing one of its resources makes three *types* of Shell widget. The three types of Shell are:

1. Application Shell. This is the main window of an application. It is the first one displayed when the application runs.

2. Top Level Shell. A window which remains visible when the Application Shell is iconified and can be iconified independently.

3. Dialog Shell. A window which cannot be iconified independently of the Application Shell. This is usually used for sub-dialogs in an application.

Far more detail concerning the Shell types is given in “Shell Types” on page 73.

Whenever you start a new design, all icons except the Shells are disabled.

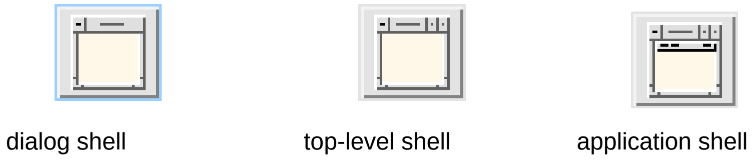


FIGURE 2-4 Shell icons

- **Click on the Application Shell icon.**

A copy of the Shell icon appears in the construction area.

Note – Because the tutorial aims to teach you how to use Sun WorkShop Visual from scratch, you are asked to start in this way. You can, however, use the Sun WorkShop Visual utility AppGuru to start an application. See Chapter 13, “Design Tools”, starting on page 417, for more details.

Widget Names

When an instance of a widget is added to the hierarchy, it is assigned two names by Sun WorkShop Visual: a *widget name* and a *variable name*. The variable name is the name by which the widget is referenced in the code. The widget name is the name used by the toolkits to assign resources. By default these names are identical. Sun WorkShop Visual tries to assign sensible default names. These are of the form *widget_nameN*, where *widget_name* reflects the class of the widget (button, form, shell etc.) and *N* is an integer assigned by Sun WorkShop Visual and which increments within the widget class (button1, button2, form1, form2 etc.) However, because several features of Sun WorkShop Visual require explicit variable names, it is a good habit to assign explicit variable names to the most important widgets as you add them.

Naming Widgets

To name the Shell widget:

1. **Double-click in the box opposite "Variable Name" at the top of the screen.**

   When you double-click in the box, all text in it is highlighted. Entering new text replaces the highlighted text.

2. **Type: `myFirstShell`**

   The name is automatically assigned to the widget when you create and select another widget. You may also assign the name by pressing <*Return*> in the variable name box.

Note – The variable name must be unique because it is used to refer to the widget structure for that instance of a widget when Sun WorkShop Visual generates code. Sun WorkShop Visual does not let you enter a variable name used elsewhere in your design. To avoid problems in compiling, never use the names of your application functions or variables, Motif or X defines or routine names, or C or C++ reserved words as widget variable names.

When you change the variable name, Sun WorkShop Visual automatically assigns the same name to the widget name unless you also explicitly specify a widget name in the "Widget Name" box.

The widget name does not have to be unique. Widgets with the same widget name can be configured to share resource settings. It is often convenient to group widgets by a common widget name so that end users can reset their resources with a single operation. You can do this by selecting all the widgets you wish to share a widget name and typing the name into the field labelled "Widget Name". "Multiple Selection" on page 23 gives more information on selecting more than one widget. The subject of shared resources is discussed more extensively in "Shared Resource Values" on page 237.

Adding Children to the Hierarchy

The Shell widget can have any kind of widget as its child; it can, however, only have one child. Therefore, you should choose a widget which can contain the rest of your layout. A MainWindow, BulletinBoard, Form, or DialogTemplate are commonly used. The DialogTemplate provides a convenient layout for the tutorial interface.

- **Click on the DialogTemplate icon.**

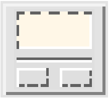


FIGURE 2-5 Dialog Template Widget Icon

If you need help identifying the icons, turn on both names and icons from Sun WorkShop Visual's Palette menu or bring up the Palette Icons Dialog from the Help Menu (<*Meta H*> <*p*>). Also note that the status line at the bottom of the Sun WorkShop Visual screen will show the name of the widget when the cursor is positioned over an icon in the palette.

See "The Palette Menu" on page 723 and "Palette Icons..." on page 739.

The DialogTemplate

The DialogTemplate is a container widget which can have three kinds of children: a MenuBar, any number of buttons and one additional child which is called the *work area*.

The DialogTemplate positions the MenuBar child at the top of the window and arranges all children which are buttons of any type in an evenly spaced row at the bottom of the window. This row of buttons is called the *button box*.

The DialogTemplate places the work area between the MenuBar and the button box, separated from the button box by a Separator (a horizontal line). The Separator is created as part of the DialogTemplate and will appear in your widget hierarchy automatically.

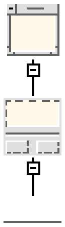


FIGURE 2-6 The DialogTemplate in the Hierarchy

In the tutorial interface, Figure 2-1, the menu bar cascade buttons and the pushbuttons which make up the button box are labeled. The work area contains the radio box, row column array and toggle buttons.

- **Assign the variable name: `myDiag` to the DialogTemplate.**

As you add widgets to the design, we recommend that you continue to assign variable names to them. Explicit names make it easier to identify widgets and are required for certain operations. However, since Sun WorkShop Visual does not strictly require names unless you refer to the widget in some way, these instructions only include this step for names which the tutorial uses later.

Dynamic Display of Layout

When you added the DialogTemplate widget, you may have noticed that your layout became visible as a small rectangle over the construction area. This is the *dynamic display window* in which Sun WorkShop Visual builds a working example of your interface.

What you see in the dynamic display is a collection of widget instances. Sun WorkShop Visual does not draw pictures of widgets but actually creates them using the same Motif function calls that your interface will use when it is running. Right now the dynamic display window has few identifiable features because it contains only the Shell and DialogTemplate.

As you add widgets and move them around, they appear in this window as they will appear in your finished interface. You can use the normal window manager facilities to move the dynamic display window to a part of your screen where it does not obstruct your view of the hierarchy.

When you add widgets to your hierarchy, they may not appear in the dynamic display window where you want them. Later, you will use the Layout Editor to achieve the correct appearance.

The layout shown in a dynamic display is a fully active prototype of your interface; you can click on the buttons, pull down the menus, type text into text fields, and so on.

Currently Selected Widget

In Sun WorkShop Visual, it is possible to have one widget selected, many widgets selected or no widgets selected. If only one is selected, it is known as the *currently selected widget.* A widget must be selected before you can do anything to it, such as setting its resources, cutting and pasting, or giving it children. The selected widget is highlighted in the construction area and in the dynamic display.

Widgets that cannot legally be children of the selected widget are grayed out on the widget palette so you cannot select them.

As a rule, widgets are selected when you first add them to the hierarchy. Therefore, when you add the DialogTemplate, it is automatically selected and the next widget you add will be its child. However, widgets are not automatically selected if they cannot have children. To select a different widget, click on its icon in the construction area.

Multiple Selection

You can select more than one widget by:

- Dragging a rectangle around the widgets.
- Selecting a widget with the mouse while the Shift key is held down. This will select the widget in addition to any currently selected.

In order to add widgets in another dialog to the selection, hold the Shift key down while selecting the Shell in the window holding area in order to retain the current selection.

When there is more than one widget selected, you can perform the following operations on them:

- Set a resource - both core and widget resources. You will only be able to set those resources which are appropriate to *all* selected widgets. See “Multiple Selection and Resources” on page 59 for more details.
- Clear the selected widgets. These widgets are then removed from your design.

No Selected Widget

By clicking in a blank part of the construction area or by holding down the Shift key while clicking over each selected widget, it is possible to have *no* widgets selected. You will then only be able to perform those actions which affect the whole design.

Adding the Buttons

The buttons at the bottom of the layout can be added directly as children of the DialogTemplate, as shown in Figure 2-7.

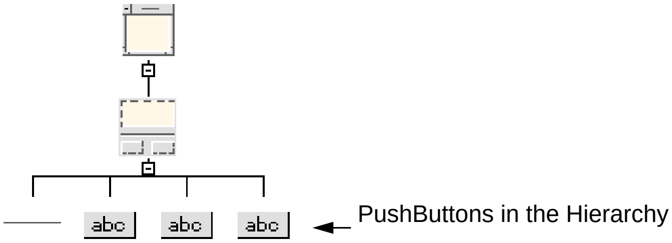


FIGURE 2-7 Hierarchy for the Buttons

- **With the DialogTemplate widget selected in the construction area, click three times on the PushButton icon.**

Each time you click, a PushButton is added as a child of the DialogTemplate. The PushButtons also appear in your dynamic display with the default label *button<n>*. Later, in *Chapter 3, "Resources", starting on page 53*, you will assign proper text strings to these labels.

The dynamic display now looks like Figure 2-8.

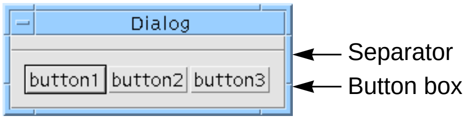


FIGURE 2-8 Dynamic Display of the Buttons

The window title "Dialog" is the default used by Sun WorkShop Visual. "Setting Resources for the Main Shell" on page 75 describes how to set your own title.

Building the Menu Bar

A menu bar at the top of the screen is a common feature of many computer interfaces. Motif provides a MenuBar widget, which is invisible until you add a series of other widgets to form pulldown menus. Sun WorkShop Visual guides you through the process of building a menu bar by graying out all widgets except the relevant ones. The hierarchy you need to add is shown in Figure 2-9.

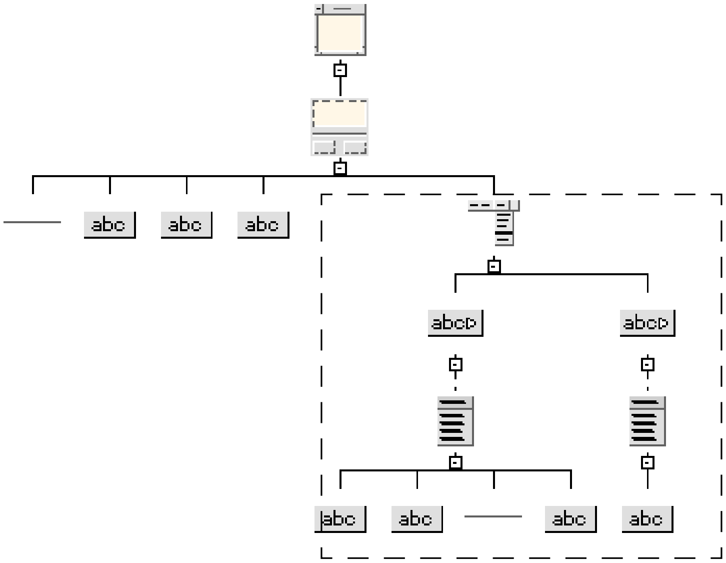


FIGURE 2-9 Hierarchy for the MenuBar and Its Children

Creating the Menu Bar

To build the menu bar:

1. **With the DialogTemplate widget selected, click on the MenuBar icon.**

   The DialogTemplate automatically places the menu bar above the work area in the dynamic display.

2. **Assign the variable name:** `main_menu` **to the MenuBar.**
3. **Click on the CascadeButton icon twice.**

When you add the CascadeButtons, they appear in your dynamic display with the default labels “cascade1” and “cascade2”, as shown in Figure 2-10.

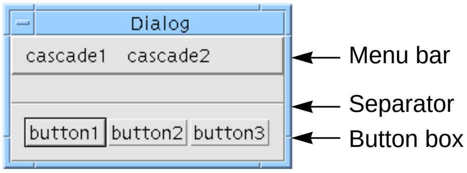


FIGURE 2-10 Dynamic Display of the CascadeButtons

You can click on these buttons with the mouse and see that they are active but they don’t do anything because they don’t as yet contain any menus.

4. **Select the first CascadeButton in the construction area and assign it the name: `procedure_cascade`**

   Note that this long name means that the second CascadeButton cannot be seen on the menubar in the dynamic display. Later in this tutorial they will be assigned more sensible names. This will make them both appear as expected. Resize the dynamic display if you wish to see both CascadeButtons now.

Adding the Menus

To attach a menu:

1. **Select the left CascadeButton in the hierarchy and click on the Menu icon.**
2. **Click on the PushButton icon twice.**
3. **Click on the Separator icon; then click on the PushButton icon again.**
4. **Click on the last PushButton and assign it the variable name: `exit_button`**

   The left cascade button in your dynamic display now has a working pulldown menu, which you can see by placing your cursor on the cascade button and holding down mouse button 1, as shown in Figure 2-11.

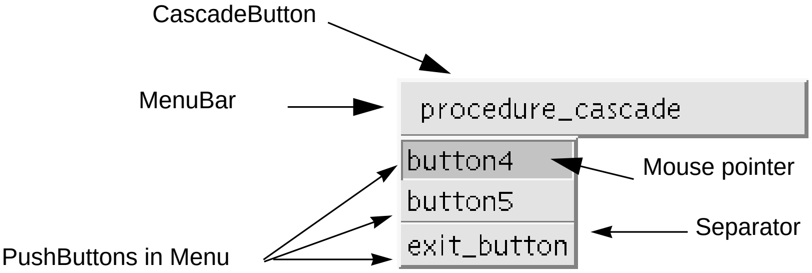


FIGURE 2-11 Pulldown Menu in the Dynamic Display

The PushButtons in this menu have default labels, which can be changed later. Menus can have three kinds of selectable children: PushButtons, ToggleButtons, or CascadeButtons (used to create submenus). They can also contain non-selectable Labels and Separators. This menu contains a Separator, which appears as a horizontal bar that separates the buttons into two regions.

To complete the MenuBar portion of the design hierarchy:

5. **Select the second CascadeButton in the construction area and assign it the variable name: `help_cascade`**
6. **Click on the Menu icon.**
7. **Click on the PushButton icon.**
8. **Click on the PushButton in the construction area and assign it the variable name: `help_button`**

The second CascadeButton now also has an active menu with one option which you can pull down with the mouse in the dynamic display.

Adding the Work Area

The interface now has a menu bar at the top and several buttons at the bottom. There is no work area until you add one. The DialogTemplate can have only one work area child. However, that child can be a container widget with multiple children. Since our work area will contain several widgets for choosing the ice cream flavors and toppings, we use a Form for the work area.

1. **Select the DialogTemplate in the hierarchy.**
2. **Click on the Form icon.**

The Form is invisible until you give it children. The options in our interface are arranged in three groupings:

- A RadioBox containing the “Double Scooper” and “Small” toggles
- A RowColumn array for the topping options
- Three ToggleButtons for the ice cream flavors

Building the Radio Box

The RadioBox, like the Form, is an invisible widget which exists only to control the behavior of its children. It can contain a group of ToggleButtons which it configures as radio buttons. Only one radio button can be selected by the user at any one time. The hierarchy you need to add is shown in Figure 2-12.

To build the radio box:

1. **Click on the Form in the hierarchy.**
2. **Click on the Frame widget icon.**

   The black line around the “Double Scooper” and “Small” radio buttons in Figure 2-1 is not the RadioBox itself but a Frame widget which contains the RadioBox. The Frame is used to display the logical grouping of the radio box components.

3. **Click on the RadioBox icon.**
4. **Click on the ToggleButton icon twice.**

   The resulting hierarchy for the radio box is shown in Figure 2-12.

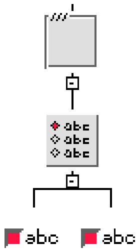
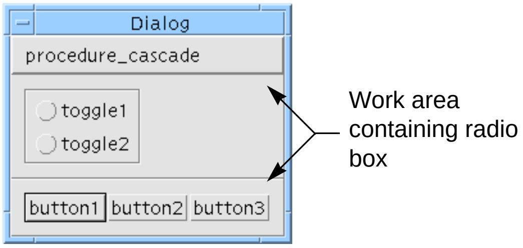


FIGURE 2-12 Hierarchy for the Framed Radio Box

The dynamic display should now resemble Figure 2-13:



Work area containing radio box

FIGURE 2-13 Dynamic Display So Far

Building the Row Column Array

A RowColumn container widget will be used for the array of labels and text fields which specify the toppings in Figure 2-1.

1. **Select the Form in the hierarchy.**
2. **Click on the RowColumn icon.**
3. **Click on icons in the following order: Label, TextField, Label, TextField, Label, TextField.**

The Label and TextField widgets must be added in this order because the RowColumn always takes its children in order when constructing rows or columns. In this case, you are building rows and each row should have a label and a text field.

The RowColumn part of the hierarchy is shown in Figure 2-14.

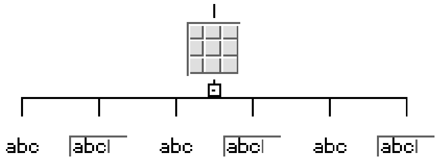


FIGURE 2-14 Partial Hierarchy: the RowColumn Widget and Its Children

This hierarchy results in the dynamic display shown in Figure 2-15.

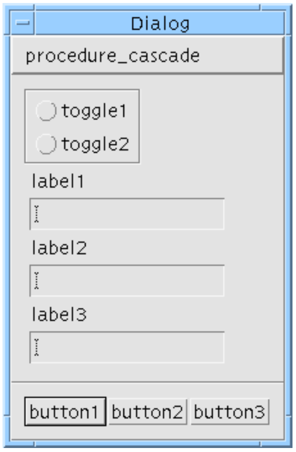


FIGURE 2-15 Dynamic Display So Far

Because, by default, the RowColumn widget is laid out in a single vertical column, it doesn't look much like the finished layout in Figure 2-1. Later in the tutorial, you will change its resources to achieve the desired effect.

Adding the Toggle Buttons

The last group of options required for the layout is the set of toggle buttons (representing ice cream flavors) at the bottom of the work area.

1. **Select the Form in the hierarchy.**
2. **Click on the ToggleButton icon three times.**

   Your design hierarchy for the tutorial interface is now complete, as shown in Figure 2-16.

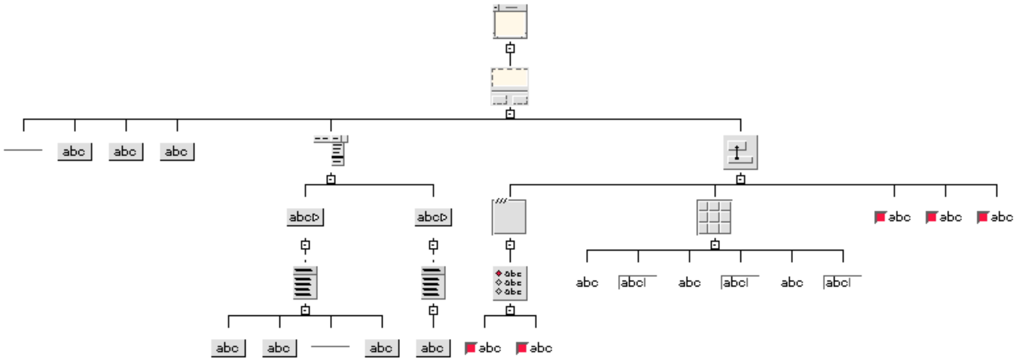


FIGURE 2-16 Complete Hierarchy

The interface, shown in Figure 2-17, is not yet finished but it contains all the necessary widgets in their proper parent-child relationships. In later chapters you will use resource panels and the Layout Editor to make it look more attractive.

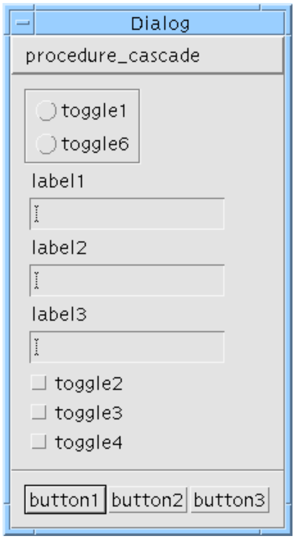


FIGURE 2-17 Dynamic Display So Far

Notice the difference between the appearance of the toggle buttons you just added and the ones in the RadioBox. When ToggleButtons are inside a RadioBox, they become radio buttons. Radio buttons are distinguished by a diamond-shaped indicator. ToggleButtons that are not the children of a RadioBox can be switched on and off independently and have a square indicator.

Now that you have added the last widget to the design, save your work using the “Save as...” command.

3. **Select “Save as...” from the File Menu.**

4. **Click to the right of the text in the “Selection” text field.**

5. **Enter a filename for your design.**

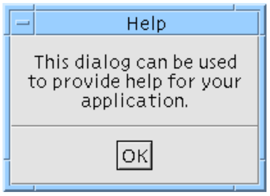
   By convention, Sun WorkShop Visual design files have the suffix `.xd`.

6. **Click on “OK”.**

   If you have already saved your design, you can use the “Save” command instead.

Adding a Window to Your Application

You have now finished setting up the main window of your interface. Most interfaces, however, have multiple windows. This chapter shows how to add a second window to your interface. The second window is a simple help screen, as shown in Figure 2-18:



This dialog can be used to provide help for your application.

FIGURE 2-18 Help Screen

This help screen will appear when the user invokes the “About This Layout” command in your interface’s Help Menu and will disappear when the user clicks on the “OK” button. This behavior is similar to that of Sun WorkShop Visual’s copyright screen. Before you start, you may want to pull down Sun WorkShop Visual’s Help Menu and select “About Sun WorkShop Visual”. Note that the copyright screen appears when you click on “About Sun WorkShop Visual” and disappears when you click on its “OK” button.

To achieve these results, you will:

1. Create an additional window for your interface.

2. Design a simple help screen within the second window.

3. Create a “Show” link to display the help screen when the “Help” command is given from the Help Menu. This happens in “Links” on page 183.
4. Create a “Hide” link to remove the help screen when the user clicks on the “OK” button. This also happens in “Links” on page 183.

Creating a Second Window

You can create a new window at any time, regardless of which widget is selected in the design hierarchy.

To add a dialog to your interface:

1. **Click on the Dialog Shell icon in the widget palette.**

   See “Shell Types” on page 73 for information on the different types of Shell.

   Sun WorkShop Visual clears the construction area and displays the hierarchy for the new window. So far, this consists only of the Shell. Note that the dynamic display for the first window is still visible. As you build the secondary window, you can see both dynamic displays at the same time.

2. **Click on the DialogTemplate icon.**
3. **Click on the Label icon.**
4. **Click on the PushButton icon.**

   The hierarchy for the subwindow and its default dynamic display are shown in Figure 2-19. Because this screen is so simple, you can use a Label instead of a container widget with children for the work area. The DialogTemplate centers the PushButton in the button box with the work area above it. There is no menu bar.

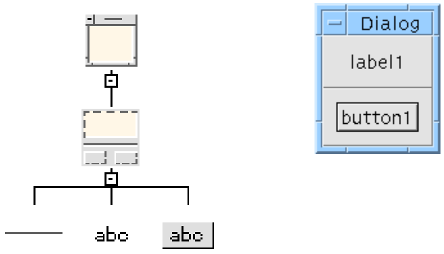


FIGURE 2-19 Hierarchy and Default Dynamic Display for Second Window

Set the text on the Label and PushButton:

5. **Double click on the Label in the design hierarchy to bring up the Label resource panel.**

6. **On the “Display” page, double-click in the “Label” box and type:**

   ```
   This dialog can be used
   to provide help for your
   application.
   ```

   Use <*Return*> to put newlines into a multi-line label. Do not put a newline at the end of the last line.

7. **Click on “Apply”.**

8. **Click on the PushButton in the design hierarchy.**

9. **Double-click in the “Label” box and type:** `OK`

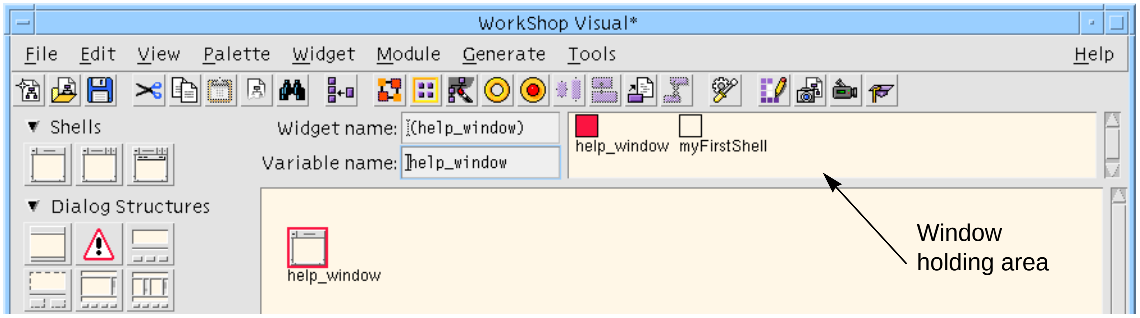
10. **Click on “Apply”.**

11. **Click on “Close”.**

Navigating Between Windows

When you add a Shell to your design, regardless of which *type* of Shell it is, a corresponding icon appears in the window holding area at the top right of the main Sun WorkShop Visual window, as shown in Figure 2-20. To move from one window’s hierarchy to another, click on the Shell icon associated with that window in the window holding area. Because most Shell icons look alike, and because icons are not necessarily shown in this area in the order in which you created them, it helps to assign explicit variable names to all Shell icons and turn on the “Show dialog names” option in the View Menu so that you can tell them apart.

The order of the Shells in the window holding area is significant. When a design is saved to a file, the Shells are saved in the order they appear in the window holding area - from left to right. When the file is loaded, Sun WorkShop Visual retains that order, displaying initially the hierarchy of the *first* (leftmost) Shell and its dynamic display. If this is not the one you wish to see first, you can change the order of the Shells in the window holding area by clicking over a Shell icon and dragging it to its new position while holding down mouse button 1.



Widget name: (help_window)

Variable name: help_window

Window holding area

FIGURE 2-20 Upper Part of Sun WorkShop Visual Screen

Assign a name to the second Shell in your design.

1. **If the Dialog Shell is not already selected in the hierarchy, select it.**

2. **Double-click in the “Variable name” field.**

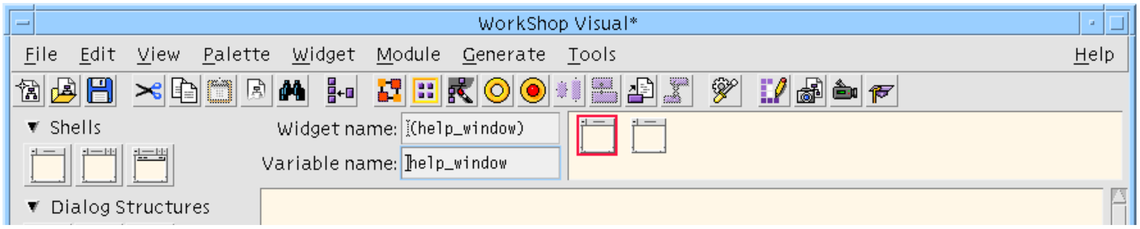
3. **Type: `help_window`**

To register the new name:

4. **Type <Return> or select any other widget in the hierarchy.**

By default, the names of the Shells are shown in the window holding area. You can turn this off, if you wish to see what type of Shell they are, by doing the following:

5. **Pull down the View Menu and select “Show dialog names” to turn the toggle off.**



Widget name: (help_window)

Variable name: help_window

FIGURE 2-21 Window Holding Area without Dialog Names

- **This concludes the step-by-step tutorial in this chapter.**

At this point, you can proceed directly to the next chapter to continue the tutorial or you can continue reading and experiment with the various editing features discussed in the following pages.

Editing the Hierarchy

Sun WorkShop Visual provides dragging, cutting and pasting facilities to let you edit the hierarchy. By using these facilities, you can alter your design dramatically without losing any of the resource values you have specified.

All editing functions act equally on the children of the selected widget. This lets you retain the relative positions of widgets inside a container widget such as a Form or RowColumn by moving the container widget and everything beneath it as a unit.

Dragging Widgets Around the Hierarchy

To drag a widget and its children to a new location, hold down mouse button 1 over the widget and drag it to its new location. When the widget is correctly positioned beneath a potential parent, a vertical line appears connecting it to the new parent. When you see the line, release the mouse button. If there is no line when you release the mouse button, the widget being dragged reverts to its former position.

Rules When Dragging Widgets

You can drag widgets to a different position beneath the same parent, or to a new parent. However, Sun WorkShop Visual does not let you drag a widget to a position which is not valid in Motif.

Widgets that are part of a composite widget, such as the ScrollBars which form part of the MainWindow, can only be dragged by dragging their parent.

Because a widget’s children are dragged with it, you cannot drag a widget to a position beneath its own child. To get this effect, use the copying facility described below.

If you change your mind after starting to drag a widget, you can cancel by dragging to an empty spot in the construction area.

The Shell widget cannot be dragged because it is not a valid child of any class of widget.

Copying Widgets

To copy a widget and its children to a new location while leaving the original widget in place, drag the widget using mouse button 2. A default variable name is assigned to the copied widget.

Edit Commands: Cut, Paste, Copy and Clear

The Edit Menu has “Cut”, “Paste”, “Copy”, and “Clear” commands which can also be used to alter the hierarchy. To copy a widget and its children onto the Sun WorkShop Visual clipboard, select the widget and use the “Copy” command (*<keypad>Copy*).

“Cut” (*<keypad>Cut*)) deletes the selected widget and its children and copies them onto the clipboard. “Clear” also deletes the selected widget and children but does not affect the clipboard. Cleared items cannot be pasted back into the hierarchy.

“Paste” (*<keypad>Paste*) inserts the contents of the clipboard directly beneath the currently selected widget. “Paste” is disabled if the clipboard is empty, or if the widget in the clipboard is not a valid child of the currently selected widget. The pasted widget is always made the last child of the selected widget. To place it in a different position, drag the selected widget with the mouse.

Copy to File, Paste from File

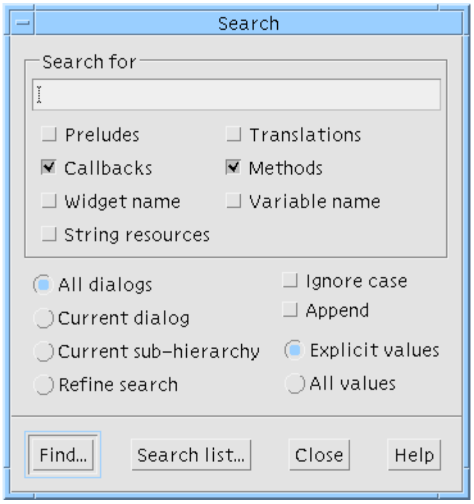
As well as copying to the Sun WorkShop Visual clipboard, you can copy a widget and its children to a clipboard file and paste in a widget from an existing clipboard file. This feature lets you build a library of design fragments, such as a standard menu bar. By convention, Sun WorkShop Visual clipboard filenames have the suffix `.cxd`.

Alternate Method of Selecting Widgets

To select any widget that is not highlighted, you can use the mouse or you can step up, down, left, or right in the hierarchy by using the arrow keys. The arrow keys only work this way when the construction area has the input focus. If the arrow keys seem to be disabled, use the *<Tab>* key to cycle the focus around the various areas of the Sun WorkShop Visual screen until they become active.

Search

The search facility, available from the Edit menu, allows you to search for strings in preludes, callbacks, methods, translations, widget and/or variable names and string resources. The Search dialog is shown in Figure 2-22.



Search for

□ Preludes □ Translations

☑ Callbacks ☑ Methods

□ Widget name □ Variable name

□ String resources

● All dialogs □ Ignore case

○ Current dialog □ Append

○ Current sub-hierarchy ● Explicit values

○ Refine search ○ All values

FIGURE 2-22 The Search Dialog

There are four main areas in the Search dialog - the text box containing the string to be found, a set of toggles to specify where to look for the string, a set of toggles to define which widgets to search and some options affecting the search.

String to be Found

You may type any string or part of a string in this text box. If you leave the text box blank, every string is matched. The search mechanism will look for strings according to which options have been selected from the String Type Panel.

String Type Panel

This area consists of a series of toggles relating to the types of string which can be set for a widget. These are preludes, callbacks, methods, translations, widget names, variable names and string resources. You may select any number of these at once. Selecting none results in no matches.

Where to Search

You can specify one of the following:

- *All dialogs* – Look through all dialogs in the current design
- *Current dialog* – Only search through the current dialog
- *Current sub-hierarchy* – Only search through the hierarchy below and including the selected widget
- *Refine search* – Only search through those widgets which matched in the previous search

Search Options

You can choose whether you wish Sun WorkShop Visual to ignore the case of letters in the string when looking for a match. You can also select whether you wish to "Append" to an existing list of widgets which were found as the result of a previous search. If you are searching string resources or widget or variable names, you may select whether you want to search only those values which you have explicitly set or all values including defaults.

Find or Search List

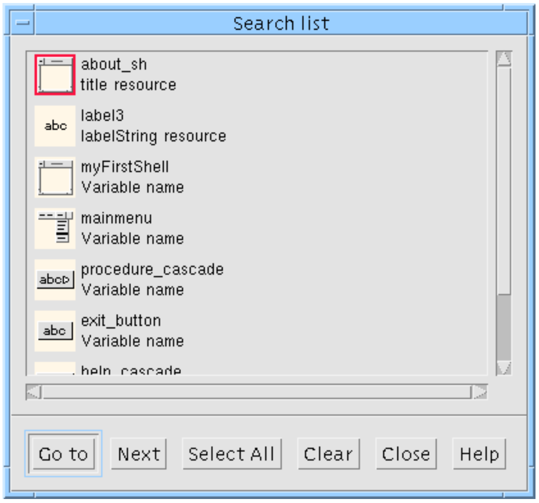
Pressing Find displays a list of widgets which match the search criteria in a separate dialog, the Search list dialog. Pressing "Search list" displays the list of widgets which have already been found - it does not repeat the search. This is useful if you have closed the Search list dialog and wish to view the same list again.

The Search List Dialog

The Search list dialog shows a list of widgets which match one or more of the search criteria. After selecting a widget from this list, the following options are available:

- *Go to* – The corresponding widget in the design hierarchy is selected. If the widget is in a part of the hierarchy which was folded, it is unfolded. Similarly, if the widget is not in the current dialog, the relevant dialog is selected first. If the string is found anywhere other than in the widget or variable name, the dialog or resource panel containing the string is opened (Callback Methods dialog, for example). Note that *Double-clicking* on an item is the same as pressing *Go to*
- *Next* – The next widget in the list is selected and the "Go to" action is invoked on the selection.
- *Clear* – Clears the list so that you can perform another search

Deleting a widget will remove it from the list, as will temporary deletions such as reset or cut and paste.



about_sh
title resource

label3
labelString resource

myFirstShell
Variable name

mainmenu
Variable name

procedure_cascade
Variable name

exit_button
Variable name

help_cascade

FIGURE 2-23 The Search List Dialog

Fast Find

If you have a complex design, finding a particular widget in the design area can be difficult. Sun WorkShop Visual allows you to go straight to a widget via the dynamic display. With the pointer over the required widget in the dynamic display, simply press <*Ctrl-G*>. Sun WorkShop Visual immediately displays the widget, unfolding the hierarchy if necessary.

Figure 2-24 shows a simple example. With the Shell keyboard focus set to “Pointer” (as explained in “Focus Policy and Fast Find” on page 41), pressing <*Ctrl-G*> over the label in the dynamic display highlights the corresponding label in the hierarchy.

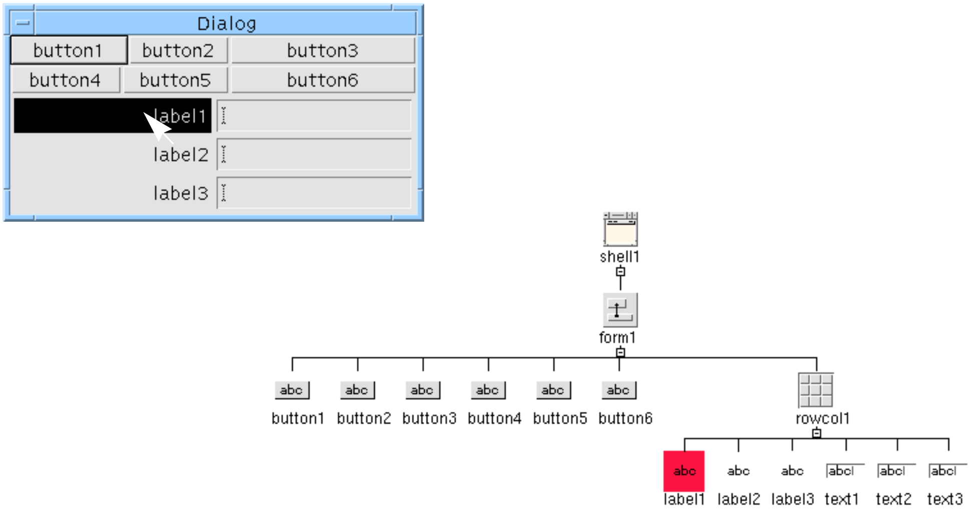


FIGURE 2-24 Fast Find

The key sequence *<Ctrl-G>* is a *translation* and can be altered using the following resource:

```
visu.fastFindTranslation: Ctrl<Key>G
```

Fast find does not have any effect on translations in the generated code.

Focus Policy and Fast Find

The fast find feature finds the widget which has the focus in the dynamic display. In order for this to work on widgets such as Labels, which do not usually receive the focus, you may need to set the focusPolicy of your Shell to “Pointer”. Do this by going to the Settings page of the Shell widget’s resource panel.

Note – Remember to set the shell’s focusPolicy back again before generating code, otherwise the final application may be confusing if there are different focus policies for different shells.

Configuring Fast Find

Ctrl-G is the translation provided by default. This is known not to conflict with any translation in the Motif widget set. Fast find works with third party widgets which support translations. For some widgets, however, this default translation may not be suitable. You may change the translation for:

1. All widgets in Sun WorkShop Visual. The following resource affects all widgets:
   ```
   visu.fastFindTranslation: Ctrl<Key>G
   ```
2. All widgets of a specified class. The following are examples which would affect all widgets of the class “XmText” in the first example and “xrtTable” in the second:
   ```
   visu.XmText.fastFindTranslation:   Ctrl<Key>F
   visu.xrtTable.fastFindTranslation: Meta<Key>M
   ```
3. Particular widget instances. The following is an example which would affect only the widget named “my_text_widget”:
   ```
   visu.my_text_widget.fastFindTranslation: Ctrl<Key>K
   ```

However, you may need or want to change the translation sequence for a particular widget type if the suggested default conflicts with the behaviour of the given widget.

Fast find translations can be configured for a widget class by specifying the input sequence required for the class. For example:

```
visu.XmText.fastFindTranslation:   Ctrl<Key>F
visu.xrtTable.fastFindTranslation: Meta<Key>M
```

Fast find can also be configured for a widget instance:

```
visu.my_text_widget.fastFindTranslation: Ctrl<Key>K
```

Note – It is possible to accidentally disable the fast find facility by providing translations of your own for a widget which override or replace the Sun WorkShop Visual fast find translation.

Disabling Fast Find

The reserved value of <None> disables the application of the fast find translation for widgets of a specified class or instance. You may wish to use this instead of altering the values of the widget in visu_config if you find that Sun WorkShop Visual's fast find mechanisms do not interact well with a given widget or widget class. Here are two examples. The first refers to a class of widgets and the second refers to a particular widget instance:

```
visu.xintGraphObject.fastFindTranslation: <None>
visu.my_text_widget.fastFindTranslation:  <None>
```

The following simple entry will disable fast find for *all* widgets in Sun WorkShop Visual:

```
visu.fastFindTranslation: <None>
```

Third party widgets can have the fast find feature disabled by using visu_config; simply set the “Disable Find Widget” toggle on the widget page.

Gadgets and Fast Find

The fast find facility uses translations, this means that gadgets (which neither support translations nor have their own window) cannot be found. In such a case, Sun WorkShop Visual will take you to the nearest ancestor of the gadget, in the hierarchy, which does support translations.

Display Options

There are a number of ways in which you can affect the display of the Sun WorkShop Visual hierarchy. Most of these are available from the View Menu. These options only change the appearance of the Sun WorkShop Visual display and do not affect your design.

Show Widget Names

This View menu option (*<Ctrl-W>*) displays the name of each widget beneath its icon in the construction area as shown in Figure 2-25. The name shown is the unique variable name assigned to the widget, not the widget name.

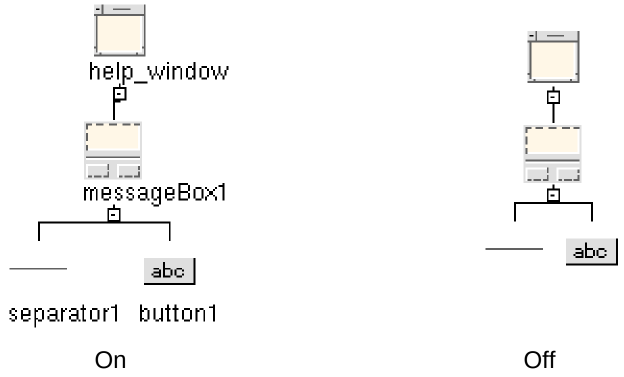


FIGURE 2-25 Show Widget Names

Show Dialog Names

Each Shell widget in the design is represented by an icon in the rectangular area at the top right corner of the Sun WorkShop Visual screen. This rectangular area is called the *window holding area.* “Show Dialog Names” (<*Ctrl-D*>), available from the View menu, displays the variable name of each Shell widget beneath its icon in the window holding area, as shown in Figure 2-26. The icon shrinks to accommodate the name. This feature is useful in layouts with multiple windows.

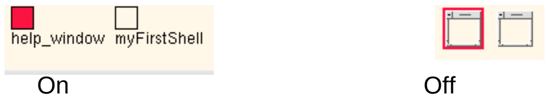


FIGURE 2-26 Show Dialog Names (Window Holding Area Shown)

Left Justify Tree

This View menu option (<*Ctrl-L*>) changes the appearance of the hierarchy in the construction area from a centered tree with branches spreading in both directions to a left-justified tree with branches spreading to the right, as shown in Figure 2-27. This feature can be useful for the rapid location of parent widgets in large designs

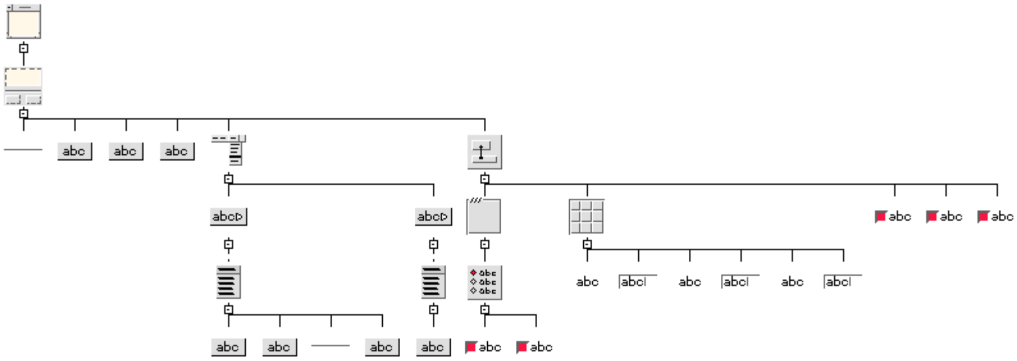


FIGURE 2-27 Left Justified Hierarchy

Shrink Widgets

This View menu option is useful when the hierarchy is large and you want to see more of the structure in the same size window. The widgets shrink to a uniform small square so that more fit in the construction area, as shown in Figure 2-28. However, the distinction between widget classes and between folded and unfolded widgets, is lost. As with the other View options, your actual design is not affected.

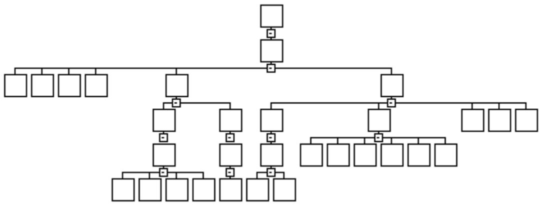


FIGURE 2-28 Shrink Widgets

Widget Annotations

Sun WorkShop Visual provides a method of annotating the design hierarchy to indicate which widgets have been given a specified attribute. The View menu contains a pullright tear-off menu labelled Annotations, as shown in Figure 2-29.

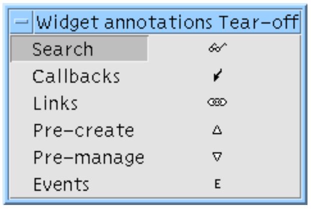


FIGURE 2-29 The Annotations Menu

There are six categories in this menu, each of which is a toggle button. To see which widgets in the design hierarchy have been given one of these attributes, set the toggle. The corresponding symbol is instantly placed next to each widget in the design hierarchy which matches the criteria of the associated symbol:

- For Callbacks, Pre-create preludes and Pre-manage preludes the criterion is that the widget has been given one of these. Selecting Callbacks also annotates any widget with a method declaration and annotates any widget which is a class where a method has been added for any descendent children
- For Links, the criterion is that the widget is the source of a link
- For Search, the criterion is that the widget was found in a previous search, as described in “Search” on page 38

Figure 2-30 shows an annotated hierarchy.

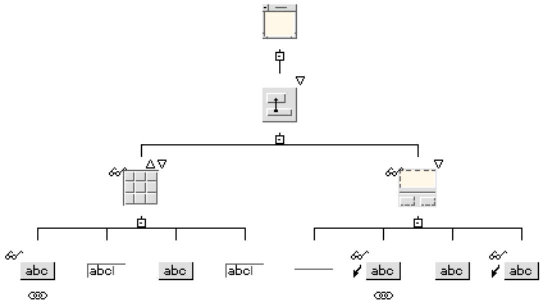


FIGURE 2-30 An Annotated Hierarchy

Configuring the Annotation Symbols

The annotation symbols are arranged around the widget icon in the hierarchy in such a way that all six symbols can be seen clearly. Where they appear in relation to the icon, how much space they use and the name of the pixmap are all specified in the Sun WorkShop Visual resource file. You can change these. See Appendix D, "Application Defaults", starting on page 867 for more details on Sun WorkShop Visual's application resources. The relevant lines are as follows - the example here is the search symbol:

```
visu*annotate_search.annotatePosition:NorthWest
visu*annotate_search.annotateWidth:10
visu*annotate_search.annotateHeight:3
visu*annotateSearchPixmap:an_search.xpm
```

The first line above specifies the geographical location NorthWest. This is in relation to the widget icon and can be any of the eight primary or secondary compass points.

Structure Colors

The "Structure colors" option in the View Menu is useful when you are building a design that uses the structured code generation features. This option color-codes widgets that are designated as function or data structures, C++ classes, or Children Only place holders.

"Structure colors" has a pullright submenu. Select "Show colors" on the submenu to display your structures in the appropriate colors. Click on the dashed line at the top of the submenu to tear it off as a reference to the color code.

Widgets that are not designated as any kind of structure are displayed against the usual background color.

Fold / unfold Widget

"Fold / unfold" is available from the Widget menu but it affects the appearance of the widgets in the construction area. As your hierarchy becomes larger, you may want to *fold* a widget which has a number of children so that its children do not take up so much space in the construction area. For example, in the tutorial layout, you may want to fold the MenuBar widget because its children fill so much display space. When a widget is folded, its children are not shown in the hierarchy. Folding widgets is only a display convenience and does not remove widgets from your design.

There are two ways of folding a widget. One involves using the Widget menu, the other involves selecting a special icon in the hierarchy.

Using the Widget Menu

Select the widget to be folded. Pull down the Widget Menu and select "Fold / unfold" (*<Ctrl-F>*). The same command unfolds the selected widget if it is already folded.

Using the Hierarchy Fold Icon

Select the fold icon in the hierarchy. This icon appears below each widget which may have children and looks like a small box containing a minus sign (-). Clicking on this icon folds the hierarchy below it.

When the hierarchy is folded, whether by this method or by using the Widget menu, the fold icon changes to display a plus sign (+). Selecting this icon will unfold the hierarchy below it.

Figure 2-31 shows a folded widget.

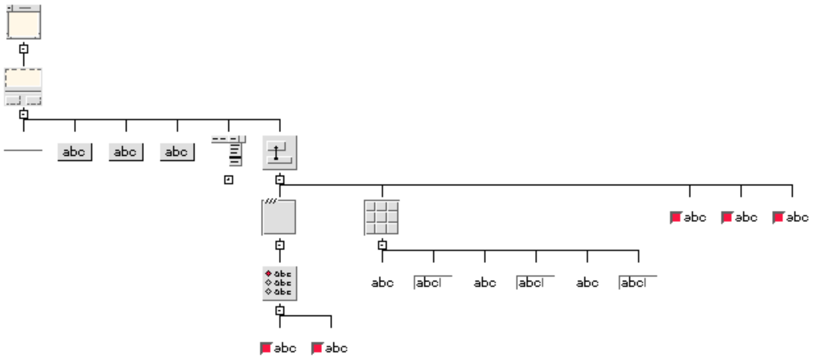
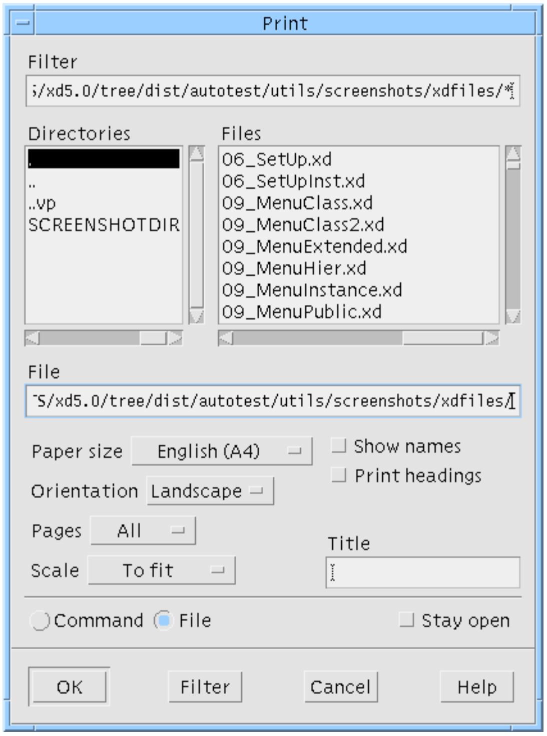


FIGURE 2-31 Tutorial Hierarchy So Far, with MenuBar Widget Folded

Printing Your Hierarchy

The Print Dialog lets you print out a hard copy of your hierarchy at any time while you are developing it. The Print Dialog is shown in Figure 2-32.



Filter

;/xd5.0/tree/dist/autotest/utils/screenshots/xdfiles/*

Directories

.
..
..vp
SCREENSHOTDIR

Files

06_SetUp.xd
06_SetUpInst.xd
09_MenuClass.xd
09_MenuClass2.xd
09_MenuExtended.xd
09_MenuHier.xd
09_MenuInstance.xd
09_MenuPublic.xd

File

~S/xd5.0/tree/dist/autotest/utils/screenshots/xdfiles/

Paper size English (A4)

Orientation Landscape

Pages All

Scale To fit

☐ Show names

☐ Print headings

Title

○ Command ● File

☐ Stay open

FIGURE 2-32 Print Dialog

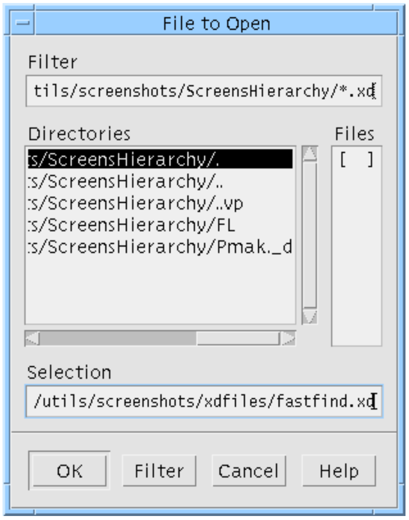
To print to a file, click on the “File” toggle and enter the filename in the text box under “File”. To send to a printer, click on the “Command” toggle and enter the command, such as `lpr`, in the same text box, which is now labelled “Command”. The output is Postscript, so a Postscript printer or viewer is required.

The option menus in the Print Dialog let you specify the page size, orientation, pages and scale. In the “Scale” option menu, the reduced scale option prints the diagram two-thirds of its actual size. Note that if the “Scale to fit” option is not selected, the diagram prints on as many pages as required. The “Pages” option menu lets you print either all the hierarchies in your design if your design contains more than one window or just the hierarchy currently displayed in the construction area.

Selecting the “Show names” toggle lets you print the variable names of the widgets. Selecting the “Print headings” toggle puts a border around the hierarchy and prints a title, which you can specify in the “Title” text field. The title is restricted to one line of text.

Using the File Browser

The file browser, shown in Figure 2-33, lets you specify the name of a Sun WorkShop Visual file to open or save. The file browser is displayed when you select any command that requires you to specify a filename, such as the “Open” and “Save as...” commands from the File Menu. You can either enter a pathname in the “Selection” field or use the mouse to select an existing filename from the “Files” list.



Filter
tils/screenshots/ScreensHierarchy/*.xd

Directories
s/ScreensHierarchy/.
s/ScreensHierarchy/..
s/ScreensHierarchy/..vp
s/ScreensHierarchy/FL
s/ScreensHierarchy/Pmak._d

Files
[ ]

Selection
/utils/screenshots/xdfiles/fastfind.xd

FIGURE 2-33 The File Browser

The “Filter” text field displays the current directory and a filename pattern to be matched in the “Files” list. You can change the current directory and filename pattern by editing the text in the “Filter” text field and clicking on the “Filter” button at the bottom of the screen.

The subdirectories of the current directory appear in the “Directories” box. To navigate through the directory structure, either click on a selection in this list and then click on “Filter”, or double-click on the selection.

The filename pattern controls the “Files” listing. Any filenames in the current directory that match the pattern appear in the “Files” box. You can change the pattern by editing the text and clicking on the “Filter” button at the bottom of the screen. If the pattern is an asterisk (*), all files in the current directory are listed. If the pattern is `*.xd`, only files that have the `.xd` suffix are listed. To select a file, either click on the filename then click on the “OK” button at the bottom of the screen, or double-click on the filename. When you select a file, Sun WorkShop Visual proceeds with the operation you requested, such as “Open”, “Read”, or “Save as....”.

When you save a file or generate code, you can either select an existing filename or specify a new filename in the “Selection” field and click on “OK”.

Note – Note that if files have been added to the current directory since the filter has been applied, they will not appear in the “Files” listing until the filter is re-applied. This is the case even if the dialog was closed when the files were added.

CHAPTER 3

Resources

Introduction

In Motif, the appearance and behavior of a widget is controlled by its resources. Resources include colors, fonts, images and text, titles, positions and sizes of windows or widgets, callbacks, and all other customizable parameters that can affect the behavior of the interface. Resources can have indirect as well as direct effects. For example, changing the label on a PushButton also changes the size of the button to allow space for the new label.

When you add a widget to the design hierarchy, Sun WorkShop Visual sets default values for all of that widget's resources. In most cases, however, you must set some resources explicitly to make the widget useful. To make it easy to set these resources, Sun WorkShop Visual groups resources on dialogs called *resource panels*.

There are two types of resource panel - "Widget" and "Core". The Widget resource panel contains resources relevant to the class of the selected widget. The Core resource panel groups together resources which apply to all classes of widget because they apply to the *base* classes - Core, Primitive and Manager. All widget classes are derived from the Core class and most are also derived from either the Primitive or the Manager class. The Core resource panel is described in "Core Resource Panel" on page 77.

When you generate source code for your design, you can also choose to generate an X resource file. This is a separate file containing resource settings which may be altered by the end user. See "Setting up the X Resource File" on page 215 for details on resource files and their generation. As well as allowing you to control which resources should be generated into the resource file, Sun WorkShop Visual also provides *loose and tight bindings* which give you greater control over the way in which resources are assigned to the widgets in your design. This is described in "Resource Bindings" on page 85.

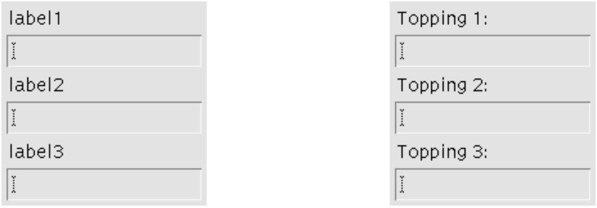
In this chapter, you will use resource panels to:

- Display the resources of widgets in the hierarchy
- Edit the text for Labels and the various types of button widgets
- Designate keyboard accelerators and mnemonics
- Designate a Help widget for the menu bar
- Set the arrangement of rows and columns in a RowColumn widget
- Edit resources for the Shell widget, including the main title at the top of the dialog frame

In addition to resource panels, Sun WorkShop Visual offers special editors for laying out widgets in a Form, setting fonts and pixmaps, editing *XmString* (Motif compound string) structures and selecting colors. See Chapter 4, “The Layout Editor”, starting on page 97 for a full description of the Layout Editor, and Chapter 5, “” for descriptions of the font, color, pixmap and compound string editors.

The Label Resource Panel

To make the tutorial interface meaningful, you must set the text of all labels and buttons to something other than the default. Begin by changing the three labels in the RowColumn widget to the text shown in Figure 3-1.



Labels with default text

After setting text

FIGURE 3-1 Labels Before and After Setting Text

First, bring up the resource panel for the first label:

1. **Double click on the first Label under the RowColumn widget.**

When you double click on the Label, Sun WorkShop Visual displays its resource panel, as shown in Figure 3-2.

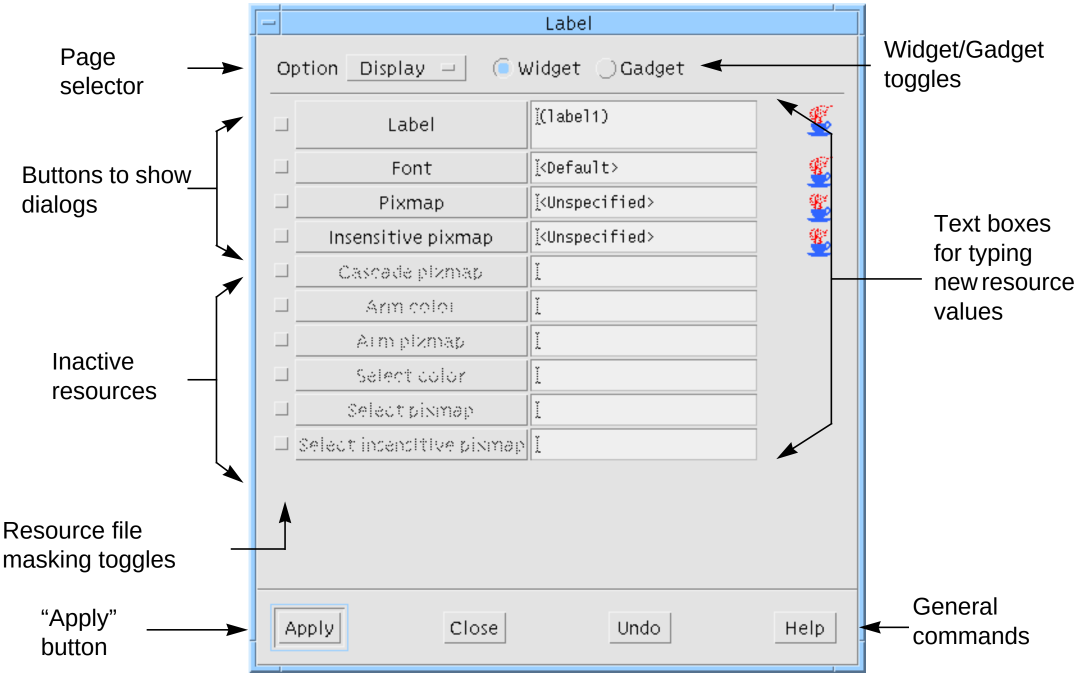


FIGURE 3-2 Label Resource Panel

Resource panels usually have several pages. The option menu in the top left corner of the panel is a page selector. If the "Display" page is not already selected:

2. **Select the "Display" page.**

Now, edit the text of the label.

3. **Double-click in the text box opposite "Label".**

Editing text in these boxes works in much the same way as assigning widget names. When you double-click, the first word in the box is highlighted. Triple-clicking highlights all the words in the box. Entering new text replaces the highlighted text.

4. **Type: `Topping 1:`**

Note – Do not press <*Return*>. Labels can contain multiple lines and pressing <*Return*> inserts a newline character into your label. If you unintentionally press <*Return*>, you can backspace to remove the newline.

---

5. **Click on the "Apply" button at the bottom of the resource panel.**

The “Apply” command sets the new resource value. When you click on “Apply”, the dynamic display shows the new label text.

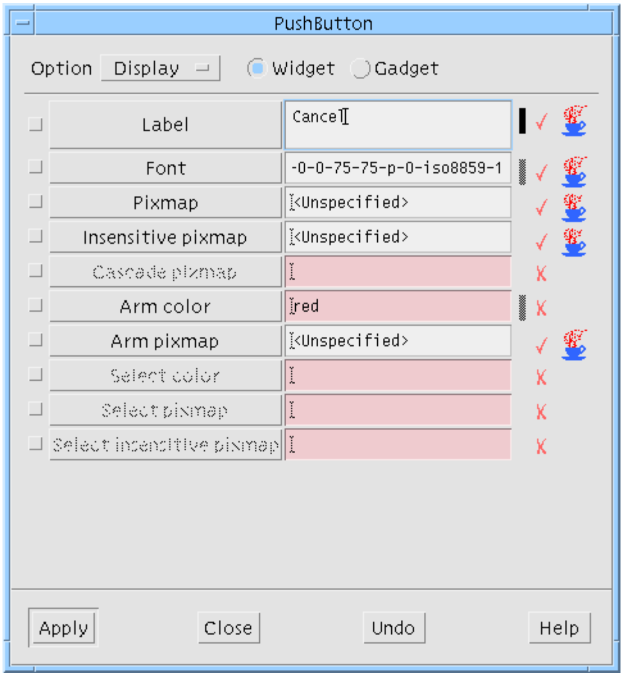
Note – If your locale is not the default “C” and you are using international text in your labels, the text will not appear correctly until you have set up the fontlist resource required for your particular language. See “Setting the Application Font Resource” on page 624 for more information. See Chapter 22, “Internationalization”, starting on page 615 if you are not sure whether this applies to you.

Regions of the Resource Panel

Although different classes of widgets have different resource panels, all resource panels have the same basic structure.

Annotations

As you change resources, a change bar appears at the far right of the resource panel, as shown next to the “Label” resource in Figure 3-3.



PushButton

Option Display ● Widget ○ Gadget

□	Label	Cancel	▌ ✓
□	Font	-0-0-75-75-p-0-iso8859-1	▌ ✓
□	Pixmap	<Unspecified>	✓
□	Insensitive pixmap	<Unspecified>	✓
□	Cascade pixmap		✗
□	Arm color	red	▌ ✗
□	Arm pixmap	<Unspecified>	✓
□	Select color		✗
□	Select pixmap		✗
□	Select insensitive pixmap		✗

FIGURE 3-3 Annotated Resources

After you press “Apply”, the change bar alters in appearance to that shown next to the “Font” resource in Figure 3-3.

Resources which will be honored in Java code have a picture of a steaming coffee cup ☕ next to them in the resource panels, as shown in Figure 3-3. The same symbol with the letter “S” over it ☕ indicates that the resource can be mapped to the property of a Swing component. For more information on using Sun WorkShop Visual for Java code generation, see Chapter 10, “Designing for Java”, starting on page 313.

Note – If Sun WorkShop Visual cannot allocate enough colors for its icons it will use the letter ‘J’ to indicate Java resources.

The tick and cross symbols shown in Figure 3-3 will only appear when running in Microsoft Windows mode. The tick indicates that the resource is applicable to Microsoft Windows and the cross indicates that it is not. This is explained more fully in “Visual Compliance Indicators” on page 363.

Resource Names

Resource names are shown on the left of a resource panel. Sometimes these names are simply labels and sometimes they are buttons which, when pressed, display a dialog relevant to the resource type. In Figure 3-2 above, all the resource names are buttons. Any resources that do not apply to the selected widget are grayed out.

Resource Values

Current resource values are shown in the boxes on the right. These boxes are either single-line text fields, multi-line text boxes or menu options. You can edit resource settings by typing into these boxes or selecting from an option menu. Default values are shown in parentheses.

Masking Toggles

The unlabeled toggle to the left of each resource name is the *resource masking toggle* which can be used at the code generation stage to mask resources in or out of the X resource file. This topic is discussed in “Masking Resources” on page 223. You do not have to set these toggles in order to set resource values.

Page Selector and Toggle Switches

The main section of the resource panel usually has several pages. The option menu at the top of the panel is a *page selector* which lets you move from one page to another. The Label panel also has a toggle switch which can be used to designate this widget as a gadget. The widget-gadget toggle appears in the resource panels for other widget classes if the widget class has a gadget counterpart. Some widget classes also have other toggle switches appropriate to the class.

General Commands: Apply

The four buttons at the bottom of the panel offer general commands. “Apply” causes your new resource settings to take effect. If you do not click “Apply”, your edited settings are lost when you select another widget or close the resource panel.

Undo, Close, Help

“Undo” makes all edited settings revert to the last applied settings. “Close” makes the resource panel disappear. “Help” displays the appropriate help screen.

Note that in Figure 3-3, the “Apply” button is highlighted. Pressing <*Return*> when a resource panel has the input focus generally executes the highlighted command. However, <*Return*> does not work in this way when you type into a multi-line text box, because the text box accepts <*Return*> as a newline character.

Now set the text of the other two Label widgets. You do not have to close the Label resource panel. However, you may need to move it if it covers the construction area.

1. **Select the second Label widget.**

   Note that the “Label” resource on the panel changes to reflect the current setting of the newly selected widget.

2. **Double-click in the “Label” box in the resource panel to highlight the default label.**

   Additional ways to edit in text boxes include: dragging with the mouse to highlight text, using <*Delete*> to delete highlighted text, or just typing at the current cursor location.

3. **Type: `Topping 2:`**

4. **Click on “Apply”.**

5. **Repeat Steps 1 through 4 for the third Label, using the text: `Topping 3:`**

6. **Click on “Close”.**

“Close” makes the resource panel disappear.

Multiple Selection and Resources

If you have more than one widget selected you can still set resources - in both the core and widget resource panels. In this case you will only be able to set those resources which are relevant to *all* the selected widgets. Sun WorkShop Visual decides which resources are relevant by finding a widget class which is common to all the selected widgets. In the case of widgets which are very different, for example a Button and a MenuBar, the only class they have in common is the low-level Core widget class. In such a case, you would only be able to change Core resources.

Resource Panels in Microsoft Windows Mode

When Sun WorkShop Visual is running in Microsoft Windows mode, it makes sure that the design you are creating is Microsoft Windows compliant. Motif resources do not map directly to Microsoft Windows resources. Some resources can be translated into something similar on Microsoft Windows (which may not necessarily be a resource), some cannot.

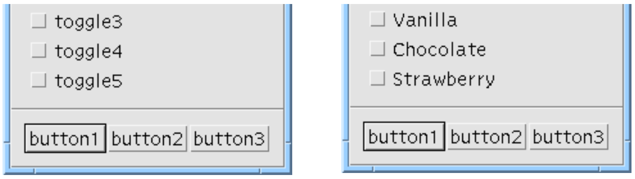
Resources are marked in the resource panels, according to their relevance to Microsoft Windows, in two ways:

- Annotation icons (tick or cross)
- Color

Resources which are not applicable to Microsoft Windows have their fields colored pink in the resource panel. You can still enter values into these fields and they will be generated for Motif, but nothing will be generated for Microsoft Windows. You can change the color of these non-Microsoft Windows resource fields by changing an entry in the Sun WorkShop Visual application resource file. See “Setting the Color of Non-Microsoft Windows Resource Fields” on page 389 for details on how to do this.

Button Widget Resources

Use the same procedure as that used for the Label widgets to set the labels of ToggleButtons, CascadeButtons and PushButtons. Set labels for the ToggleButtons in the work area as shown in Figure 3-4.



Default labels

After setting

FIGURE 3-4 ToggleButtons Before and After Setting Labels

1. **Double click on the first of the set of three ToggleButtons to bring up its resource panel.**

   You can also bring up the selected widget's resource panel by selecting the "Resources" item in the "Widget" Menu, by pressing the Resources toolbar icon or by pressing <*Return*> when the widget has the input focus in the construction area.

2. **Click twice in the "Label" box and type: `Vanilla`**
3. **Click on "Apply".**
4. **Select the second ToggleButton.**
5. **Click twice in the "Label" box and type: `Chocolate`**
6. **Click on "Apply".**
7. **Select the third ToggleButton.**
8. **Click twice in the "Label" box and type: `Strawberry`**
9. **Click on "Apply".**

Shared Resource Panels

Notice that the resource panel for a ToggleButton looks the same as for a Label. This is because, in Motif, the ToggleButton widget class is *derived* from the Label class. Another way of saying this is that the ToggleButton is a specialized kind of Label, also called a *subclass* of Label. The Label is the ToggleButton’s *superclass*.

Widget Class Inheritance

The ToggleButton class *inherits* most of its characteristics, including most of its resources, from the Label class. By means of inheritance, all Motif widget classes are organized in a hierarchy known as the *class hierarchy*. The class hierarchy is an abstract hierarchy of available widget classes and should not be confused with the design hierarchy you are building on the screen.

Several types of buttons - ToggleButtons, PushButtons, CascadeButtons and DrawnButtons - are derived from the Label class. You can set the text of all widgets of these classes using the same resource panel without having to close and re-open it.

Set labels for the radio buttons (the ToggleButtons in the RadioBox) as shown in Figure 3-5.

○ toggle1
○ toggle2
○ Large
○ Small

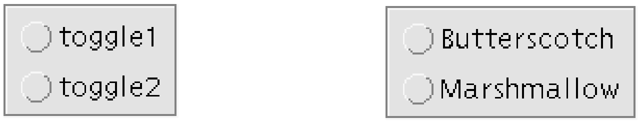
Default labels

After setting

FIGURE 3-5 Radio Buttons Before and After Setting Labels

1. **Double click on the first toggle on the RadioBox to bring up its resource panel.**
2. **Click twice in the “Label” box and type: `Large`**
3. **Click on “Apply”.**
4. **Select the second radio button.**
5. **Click twice in the “Label” box and type: `Small`**
6. **Click on “Apply”.**

If you assign a label to the radio button which is longer than the default label, the Frame widget changes size to accommodate the new width. This represents a chain reaction: the ToggleButtons resize to accommodate longer text strings, then their RadioBox parent and its Frame parent resize to fit in turn. Motif does all this automatically. This behavior is not obvious in the tutorial layout but it is illustrated in Figure 3-6.



Default labels

Longer text strings

FIGURE 3-6 Radio Buttons With Longer Labels

Use the same Label resource panel to set the label resources of the CascadeButtons as shown in Figure 3-7.

procedure_cascade
Procedures
Help

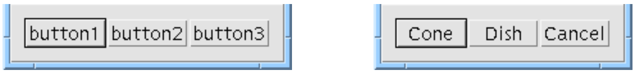
Default labels

After setting

FIGURE 3-7 Menu Bar Before and After Setting CascadeButton Labels

7. **Select the first CascadeButton and assign it the label:** **`Procedures`**
8. **Select the second CascadeButton and assign it the label:** **`Help`**

   Finally, set the labels of the three PushButtons in the button box, as shown in Figure 3-8. PushButtons also resize automatically to accommodate their labels.



Default labels

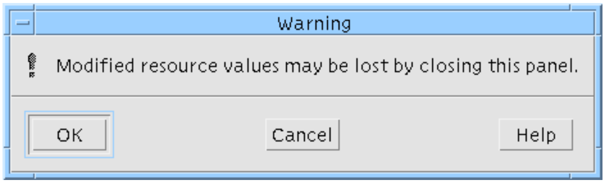
After setting

FIGURE 3-8 Button Box Before and After Setting Resources

9. **Select the first PushButton and assign it the label:** **`Cone`**
10. **Select the second PushButton and assign it the label:** **`Dish`**
11. **Select the third PushButton and assign it the label:** **`Cancel`**

Tip

Remember to click on “Apply” after setting resources, or your new settings will have no effect. If you do not apply your new settings before selecting another widget, Sun WorkShop Visual displays the warning shown in Figure 3-9.



Warning

Modified resource values may be lost by closing this panel.

FIGURE 3-9 Warning that New Resources Were Not Applied

Click “Cancel” on this display to return to the resource panel.

Resources for Menu Items

Next, set resources for the buttons in the pulldown menus of the menu bar. So far you have set only the CascadeButton labels and the pulldown menus look like Figure 3-10.

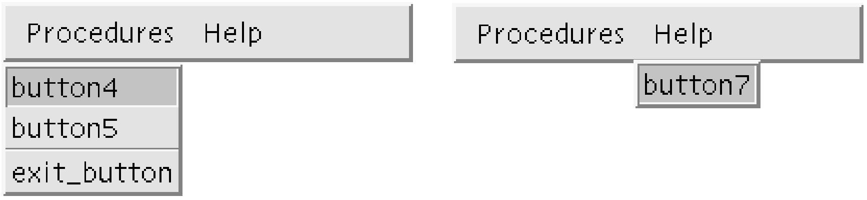


FIGURE 3-10 Pulldown Menus After Setting CascadeButton Labels

1. **Select the first PushButton child of the first Menu widget.**
2. **Change the “Label” resource to: `Wash Dishes...`**

   By convention, the ellipsis (...) is used to indicate that a menu item brings up an additional dialog box before the command is executed.
3. **Select the second PushButton and assign it the label: `Count Money`**
4. **Select the third PushButton and assign it the label: `Exit`**
5. **Select the PushButton child of the second Menu widget.**

6. **Assign it the label: `About This Layout...`**

   The pulldown menus now look like Figure 3-11.

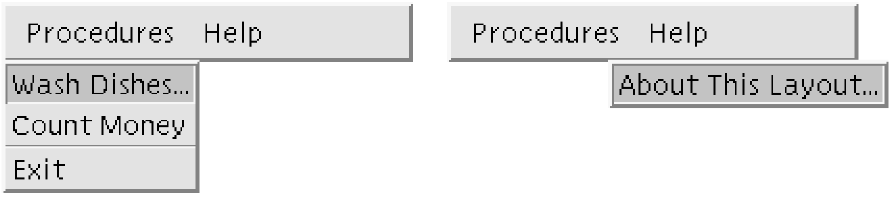


FIGURE 3-11 Pulldown Menus After Setting PushButton Labels

In the following sections, you will:

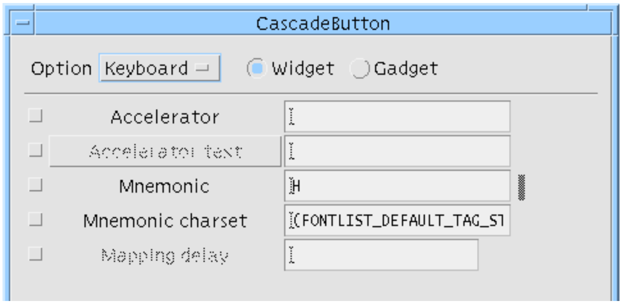
- Designate <*Ctrl-E*> as an accelerator for the "Exit" button
- Put a visible "Control+E" label on the "Exit" button to help the user remember the accelerator
- Designate "H" as a mnemonic for "Help" and "A" as a mnemonic for "About This Layout"

The Keyboard Page

Accelerators and mnemonics are keyboard resources which are found on a separate page of the Label resource panel. Use the following steps to set an "H" mnemonic on the "Help" CascadeButton:

1. **Select the "Help" CascadeButton.**
2. **Select "Keyboard" from the resource panel's page selector.**

   This brings up the "Keyboard" page, shown in Figure 3-12. The resources that are not grayed out are the ones which apply to the CascadeButton class.



CascadeButton

Option Keyboard ● Widget ○ Gadget

□ Accelerator

□ Accelerator text

□ Mnemonic: H

□ Mnemonic charset: (FONTLIST_DEFAULT_TAG_ST

□ Mapping delay

FIGURE 3-12 Keyboard Resources for the CascadeButton

Mnemonics

Sun WorkShop Visual lets you set keyboard mnemonics which work like those in the Sun WorkShop Visual interface. A mnemonic can be any character, even one which is not in the label. It is easiest for the end user if you use a character which appears in the label, preferably the first character. Mnemonics must be unique within a menu bar or menu.

1. **Double-click in the "Mnemonic" box.**
2. **Type:** `H`
3. **Click on "Apply".**
4. **Select the PushButton child of the "Help" menu.**
5. **Double-click in the "Mnemonic" box.**
6. **Type:** `A`
7. **Click on "Apply".**

Note that the "H" and "A" characters now appear underscored, which is Motif's way of indicating a mnemonic. This lets the user invoke the "About This Layout" command in two ways: with the mouse, or by pressing <*Meta-H*>, <*A*>.

You may have to reset the widget in order for the mnemonic to show in the dynamic display. Do this by checking that the Cascade Button is still selected and either pressing the Reset button on the toolbar or choosing "Reset" from the Widget menu.

Accelerators

Now add the keyboard accelerator for the Exit button. As in Sun WorkShop Visual, an accelerator immediately executes a menu command, whether or not the menu is displayed.

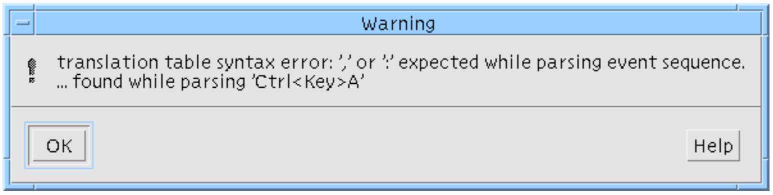
1. **Select the "Exit" button (the third PushButton under the first pulldown Menu).**

2. **Double-click in the "Accelerator" box.**

3. **Type the text string: `Ctrl<Key>E`**

4. **Click on "Apply".**

These steps make the accelerator active. When the interface is running, *<Ctrl-E>* will have the same effect as the "Exit" button. The exact syntax of the accelerator is important. If a syntax error occurs, Sun WorkShop Visual displays the error message shown in Figure 3-13 when you try to apply.



```
translation table syntax error: ',' or ':' expected while parsing event sequence.
... found while parsing 'Ctrl<Key>A'
```

FIGURE 3-13 Accelerator Syntax Error Message

Accelerator syntax is the same as that used for translation tables. This topic is discussed in "Translations and Actions" on page 190.

Accelerator Text

A related resource is *accelerator text*. This resource displays extra text to the right of a menu option to remind the user of its accelerator. Since accelerator text is just a display convenience, you do not have to use any particular syntax. "Control+E", "Ctrl-E" and "^E" are some common forms.

1. **Double-click in the "Accelerator Text" box.**

2. **Type the text string: `Control+E`**

3. **Click on "Apply".**

When you pull down the left menu, you now see the new labels on all the buttons and the designated accelerator text on the Exit button, as shown in Figure 3-14.

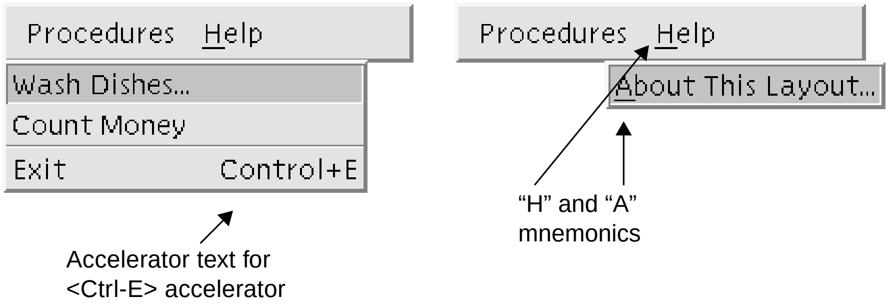


FIGURE 3-14 Pulldown Menus After Setting Resources

4. **Click on “Close”.**

Designated Help Widget

The *Motif Style Guide* suggests designating one CascadeButton child of the MenuBar as the Help widget. The Help widget always appears at the right end of the menu bar.

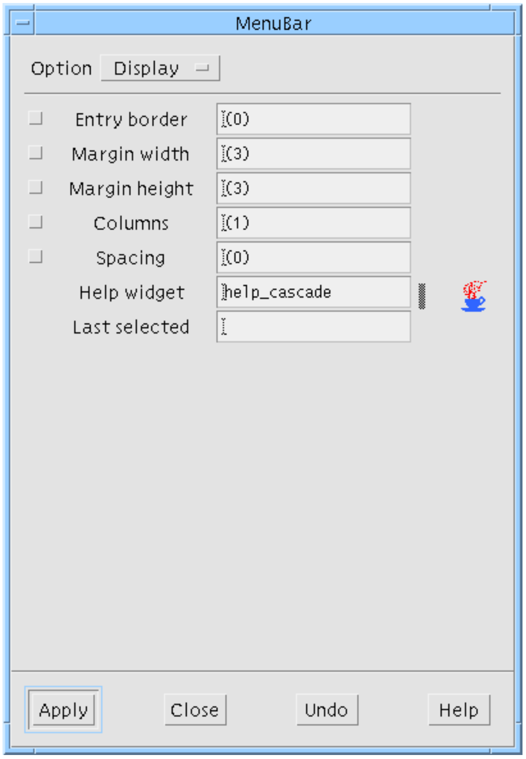
1. **Double click on the MenuBar to bring up its resource panel.**

   The MenuBar is a specially configured RowColumn and shares its resources. The resources that are not grayed out apply to the MenuBar.

2. **Select the “Display” page if not already selected.**

   To designate a Help widget:

3. **Double Click in the “Help widget” field and type: `help_cascade`**



Option Display

□ Entry border (0)

□ Margin width (3)

□ Margin height (3)

□ Columns (1)

□ Spacing (0)

Help widget help_cascade

Last selected

FIGURE 3-15 Display Resources for the MenuBar

4. Click “Apply”.

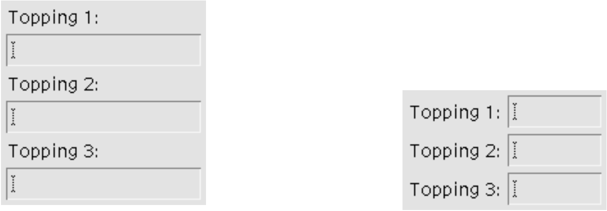
The Help widget must be one of the CascadeButtons in the menu bar. You must type the CascadeButton’s variable name exactly. If you make a mistake, Sun WorkShop Visual does not accept the entry.

The dynamic display does not show this change automatically. To see the effect of designating the Help widget, resize the dynamic display.

Leave the MenuBar resource panel open.

RowColumn Resources

By default, the RowColumn widget has one vertical column. Its resources can be set to change it from this default state to an arrangement of three horizontal rows, as shown in Figure 3-16.



RowColumn with default resources

After setting explicit resources

FIGURE 3-16 RowColumn Array, Before and After Setting Explicit Resources

To achieve the result shown in Figure 3-16, you must also set the size of the TextFields.

1. **Select the three TextField widgets in the hierarchy.**

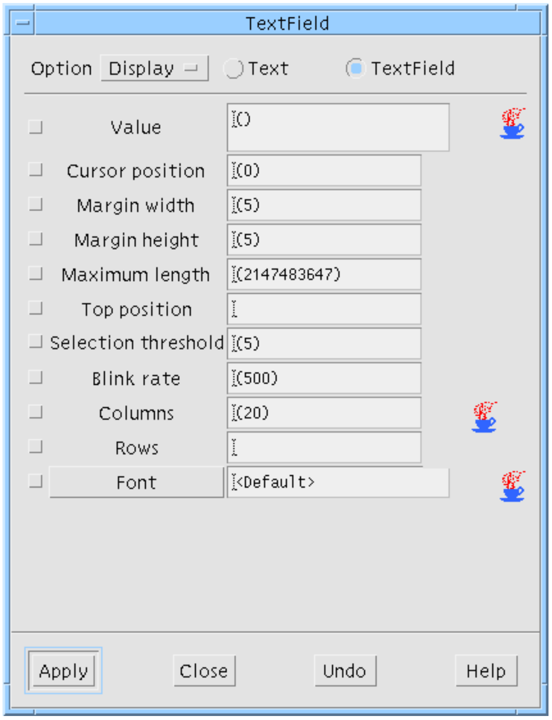
   You can do this either by dragging a rectangle around the widgets or by selecting one widget and then holding down the Shift key while selecting the other two.

   Note that when you select a TextField widget, the MenuBar resource panel, which is still present on the screen, becomes inactive. Its “Apply” button is grayed out and if you try to type into its text fields, your machine beeps. The panel becomes active again whenever a widget of the same base class (RowColumn) is selected.

2. **Bring up the TextField resource panel and select the “Display” page.**

   Invoke the resource panel by double-clicking over one of the TextFields, selecting “Resources” from the “Widget” menu or pressing the Resources toolbar icon.

   The TextField is a variant of the Text widget and shares many of its resources. The distinction is that Text widgets can contain multiple lines of text, while TextFields are limited to a single line. Because these classes have no gadget counterparts, this resource panel has no widget-gadget toggle. There is, however, a toggle to change the widget from TextField to Text. This toggle lets you change from one variant of the Text widget to the other without disturbing your hierarchy or resource settings.



TextField

Option Display ○ Text ● TextField

□ Value ()

□ Cursor position (0)

□ Margin width (5)

□ Margin height (5)

□ Maximum length (2147483647)

□ Top position

□ Selection threshold (5)

□ Blink rate (500)

□ Columns (20)

□ Rows

□ Font <Default>

FIGURE 3-17 “Display” Page of TextField Resource Panel

We want to make the text field boxes narrower. The size of the text field boxes is determined by two factors: the TextField’s “Columns” resource and the rules imposed by its RowColumn parent. Begin by setting the “Columns” resource to a smaller number:

3. **Double-click in the “Columns” box.**

4. **Type: 8**

5. **Click on “Apply”.**

   The TextFields are now narrower.

   The “Columns” resource of a TextField or Text widget only affects the size of the box and not the number of characters the user can enter. If you want to limit input to a specific number of characters, set the “Maximum length” which defaults to a very large number.

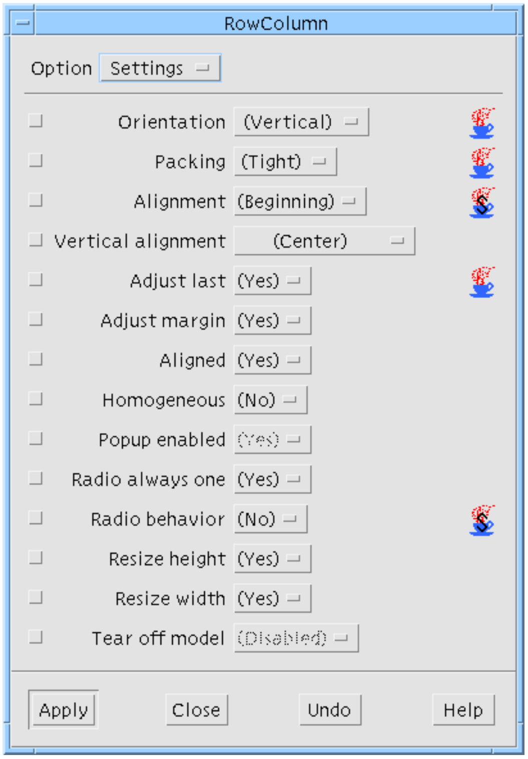
The RowColumn Resource Panel

To get the layout of three horizontal rows, you need to set the resources of the RowColumn.

1. Select the RowColumn widget in the construction area.

If you already have the MenuBar resource panel up on your screen, it becomes active again although its title is changed to “RowColumn”. If it is not on the screen, display it by double-clicking on the RowColumn again.

2. Select the “Settings” page.



Option Settings

□ Orientation (Vertical)
□ Packing (Tight)
□ Alignment (Beginning)
□ Vertical alignment (Center)
□ Adjust last (Yes)
□ Adjust margin (Yes)
□ Aligned (Yes)
□ Homogeneous (No)
□ Popup enabled (Yes)
□ Radio always one (Yes)
□ Radio behavior (No)
□ Resize height (Yes)
□ Resize width (Yes)
□ Tear off model (Disabled)

FIGURE 3-18 “Settings” Page of RowColumn Resource Panel

The “Settings” page, shown in Figure 3-18, lists resources that have a limited number of possible settings. Therefore, it has option menus instead of text fields in its right column.

3. In the “Orientation” option menu, select “Horizontal”.

4. **In the “Packing” option menu, select “Column”.**

   “Horizontal” orientation lays the RowColumn out in rows rather than columns. “Column” packing is necessary if the RowColumn is to have more than one row or column. To see the effect of these two resources:

5. **Click on “Apply”.**

   At this point, the RowColumn appears in a horizontal row, with all the cells the same size. When “Orientation” is “Horizontal”, the sense of rows and columns is reversed. Therefore, to make three rows you must set the “Columns” resource.

6. **Select the “Display” page.**

7. **Double-click in the “Columns” box and type: 3**

8. **Click on “Apply”.**

   The result is shown in Figure 3-19.

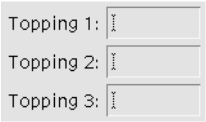


FIGURE 3-19 RowColumn Portion of the Dialog with Horizontal Orientation

Shell Resources

The Shell widget exists primarily as an interface between your application and the X window system. Its behavior is mainly controlled by the window manager and by the widgets it contains. However, it does have a few interesting resources of its own.

- **Double click on the Shell widget, myFirstShell, to bring up its resource panel.**

At the top of this panel is a toggle switch which sets the *type* of Shell widget. Although there are three Shell widgets on the widget palette, they are actually the same but with this toggle set differently.

Shell Types

Although every window starts with a Shell widget, there are different types of windows and different types of Shell widgets. Sun WorkShop Visual provides the following Shell widgets, as shown in Figure 3-20:

- *Application Shell* – The main window of the application, which is the first one displayed when the application runs
- *Top Level Shell* – A window other than the Application Shell which remains visible when the Application Shell is iconified and can be iconified independently
- *Dialog Shell* – A window which cannot be iconified independently of the Application Shell

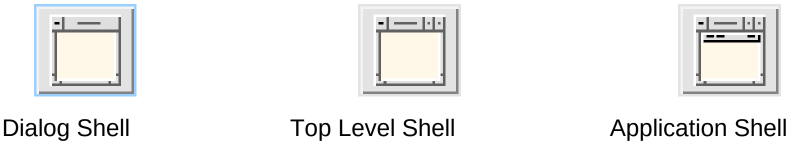


FIGURE 3-20 Shell Widgets on Palette

Note – The behavior described above applies to `mwm`. With `twm`, although Shell behavior is the same, it looks different because `twm` can turn Dialog Shells into pseudo-icons to reduce their size. The pseudo-icons are just a visual convenience for cleaning up your display. Internally, they are distinguished from true icons and they look different on your screen.

All windows in the design close when the Application Shell is closed.

Examples of Shell Types in the Sun WorkShop Visual Interface

To see some possible uses of different Shell types, look at the Sun WorkShop Visual interface itself. The main screen with the widget palette and construction area is the Application Shell. Dialogs, resource panels and the Layout Editor screen are all Dialog Shells. You cannot iconify these windows separately using the window manager.

To remove them from the display, you must close them, either using the window manager or by clicking on a “Close” button, which closes the window internally via an “Activate” callback. All open Dialog Shells disappear when the main window is iconified and reappear when it is restored.

The “Palette Icons” help panel is an example of a Top Level Shell. While it does not come up automatically when Sun WorkShop Visual starts, it can still be iconified independently once you have displayed it. Its popup subwindows, like all Dialog Shells, are children of the main Application Shell and do not close or iconify with the “Palette Icons” help panel.

Shell Type in the Dynamic Display

The Shell type is not reflected in the dynamic display. All windows created for dynamic display are really Dialog Shells and cannot be iconified independently. You can configure Sun WorkShop Visual to use Top Level shells rather than Dialog Shells - see Appendix D, “Application Defaults” for details. The generated code creates the type of Shell you specify for each window at run time.

Application Shell Requirement

You should have at least one Application Shell in each design. If you have no Application Shell in your design, the application will not display any windows. Sun WorkShop Visual shows you a warning message at code generation time if you do not have an Application Shell.

You can have more than one Application Shell in your design. In this case, the *main()* program generated by Sun WorkShop Visual creates all the Application Shells but displays only one of them. Sun WorkShop Visual cannot tell which one you want displayed first. If you have more than one Application Shell in your design, you may have to write your own *main()* program or edit the generated one to start with the correct Application Shell. To get similar results without ambiguity, use only one Application Shell for your first window, use Top Level Shells for other primary windows and use callbacks or links to display all windows but the first. Using more than one Application Shell is not recommended.

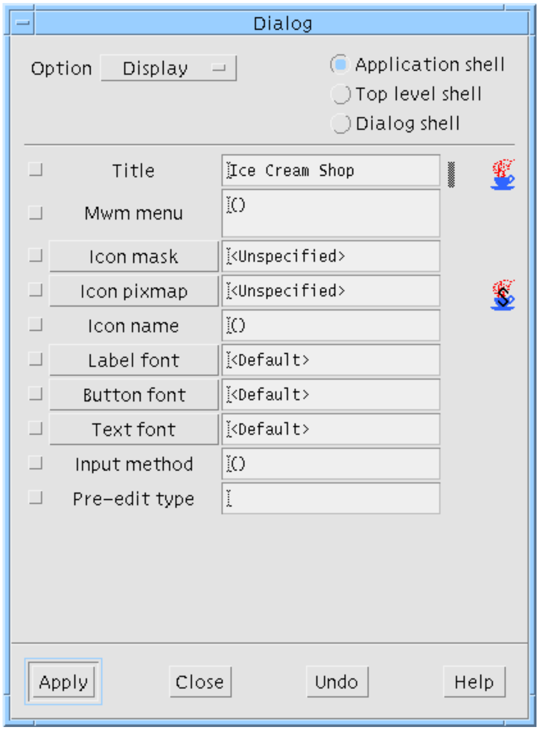
Every application must have at least one (and usually only one) *Application Shell* which serves as the main window and as the interface of the application to the X window system.

Subsidiary windows can be either *Dialog Shells* or *Top level Shells*. This distinction is explained more fully in “Shell Types” on page 73.

Setting Resources for the Main Shell

The main Shell in your design, *myFirstShell*, is already an Application Shell. You just need to change its title:

1. **Display the Shell's resource panel, if it is not already displayed.**
2. **Select the "Display" page.**
3. **Double-click in the "Title" box and type: `Ice Cream Shop`**



Option: Display

● Application shell
○ Top level shell
○ Dialog shell

Title	Ice Cream Shop
Mwm menu	()
Icon mask	<Unspecified>
Icon pixmap	<Unspecified>
Icon name	()
Label font	<Default>
Button font	<Default>
Text font	<Default>
Input method	()
Pre-edit type	

FIGURE 3-21 "Display" Page of the Dialog (Shell) Resource Panel

4. **Click on "Apply".**
5. **Click on "Close".**

The layout now looks like Figure 3-22.

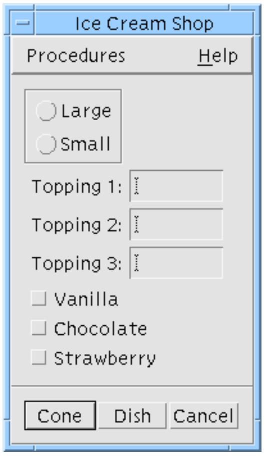


FIGURE 3-22 Tutorial Interface So Far

Setting Resources for the Secondary Window

In your application, you have another Shell for the Help window. This Shell is a Dialog Shell. You need to change the title of this window.

To view its resource panel:

1. **Double click on the Shell for the Help window.**
2. **Select the "Display" page.**
3. **Double-click in the "Title" box and type:** `Help`
4. **Click on "Apply".**
5. **Click on "Close".**

Navigating in the Resource Panels

Because Motif has so many resources, Sun WorkShop Visual uses multiple-page resource panels to display them on the screen. You may find it useful to refer to Chapter 27, "Widget Reference", starting on page 743 while you are becoming familiar with the structure of the resource panels. That chapter has a list of resources by widget class and page.

Some resources that are not available on the resource panel of a specific widget class can be found on the Core resource panel, which is discussed in “Core Resource Panel” on page 77.

Settings

In general, any multiple-choice resource - one with a limited number of settings - is found on the “Settings” page, where you can set it with an option menu. All other resources that require you to type in a new value or call an additional dialog to set the new value are divided among the other pages. Except for the “Settings” page, resources are organized loosely by topic.

Display, Margins

Resources that affect the widget’s appearance are generally found on the “Display” page. These resources include text, colors, fonts and dimensions. The Core resource panel also includes some resources that affect the dimensions and the location of the widget.

Labels and Label derivatives have enough display resources to require a second page. The Label resource panel divides these resources into a “Display” page, containing color, font, text and pixmap resources, and a “Margins” page, containing size and margin width resources.

Keyboard

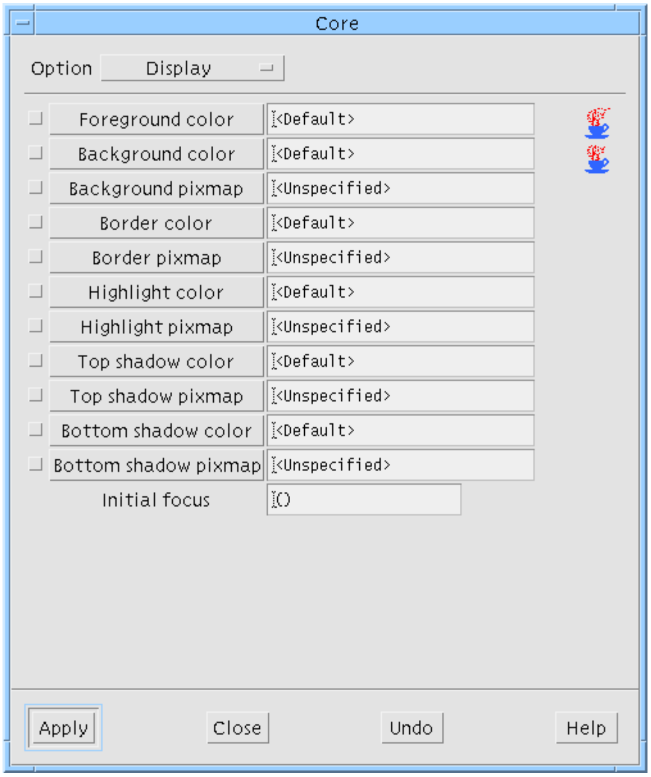
The “Keyboard” page lets you set keyboard mnemonics and accelerators for widgets that can have them.

Core Resource Panel

Sun WorkShop Visual gives you access to resources for these broad superclasses via a single resource panel, the *Core resource panel*. To bring up the Core resource panel for a specific widget:

1. **Select a widget.**
2. **Pull down the Widget Menu and select “Core resources...”.**

A page from the Core resource panel is shown in Figure 3-23.



Option Display

□	Foreground color	<Default>
□	Background color	<Default>
□	Background pixmap	<Unspecified>
□	Border color	<Default>
□	Border pixmap	<Unspecified>
□	Highlight color	<Default>
□	Highlight pixmap	<Unspecified>
□	Top shadow color	<Default>
□	Top shadow pixmap	<Unspecified>
□	Bottom shadow color	<Default>
□	Bottom shadow pixmap	<Unspecified>
	Initial focus	()

FIGURE 3-23 “Display” Page of the Core Resource Panel

Display Page of the Core Resource Panel

The “Display” page of the Core resource panel has basic color and pixmap resources. For example, you can set the colors for the widget’s foreground, background, highlighting and shadows. You will do this in Chapter 5, “Other Editors”.

Dimensions Page of the Core Resource Panel

The “Dimensions” page offers resources that affect the widget’s size and location on the screen. Class-specific dimension resources may override the settings on the Core panel. You may want to experiment with the effects of setting “Shadow thickness” on a TextField or “Highlight thickness” on a PushButton.

Settings Page of the Core Resource Panel

The “Settings” page offers miscellaneous multiple-choice settings which apply to most widget classes. This page also allows you to change the “Map when managed” setting, which works in conjunction with the managed toggle, described below.

Code Generation Page of the Core Resource Panel

The “Code Generation” page gives you increased control over the generation of code. For example, you can designate a specific widget as static, local, or global. See “Accessing Widgets in Callbacks” on page 180 for a reason to change a widget‘s access. If you are using C++, you can also designate it as private, protected, or public. “Widget Member Access Control” on page 280 discusses C++ access. The “Code Generation” page also allows you to create your own derived C++ classes. This subject is described in detail in “C++ Classes” on page 254.

Managed Toggle

The managed toggle also appears on the “Code Generation” page. By default all widgets are generated as managed, with the exception of the “Apply” button in a SelectionBox that is not a child of a Dialog Shell. This state can be modified using the “Managed” toggle in the “Code generation” page of the Core resource panel. Usually this just means that the code to manage the widget is omitted from the generated code. For widgets or gadgets that are components of composite widgets, the generated code explicitly unmanages the widget if the toggle is off, since the toolkit always creates these widgets as managed. For the “Apply” button of a Selection Box, code to explicitly manage the button is generated if the toggle is on. See “Manipulating Widgets” on page 181 for more information on widget manipulation.

Include in Resource Bindings Toggle

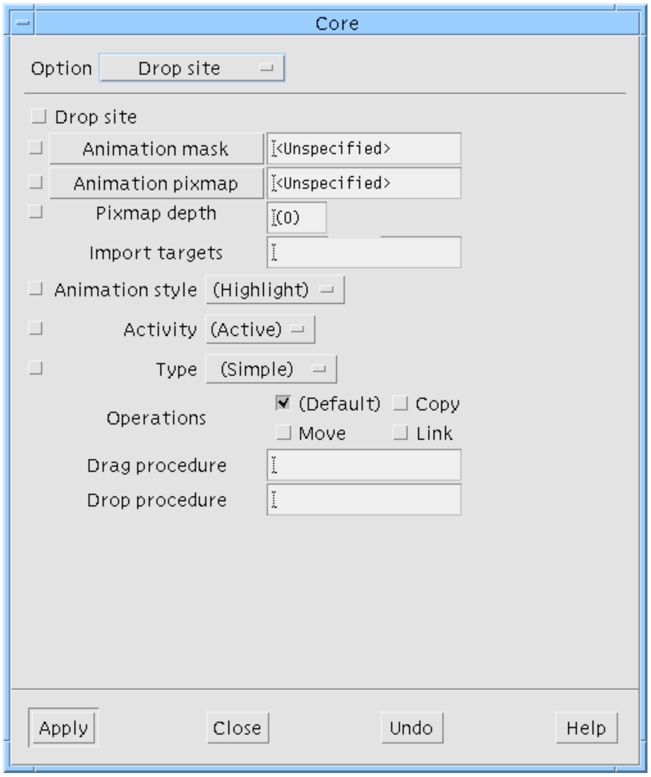
The toggle labelled “Include in Resource Bindings” refers to the generation of resources for the selected widget. This is explained in full in “Tight Bindings” on page 92.

Drop Site Page of the Core Resource Panel

This is where you can specify a *drop site.* A drop site is a widget that is prepared to receive certain types of data from other widgets by means of the drag and drop mechanism introduced in Motif 1.2. Drag and drop allows you to pass information between widgets by selecting the data and dragging it using the mouse.

The initialization of a drag is a dynamic function that would normally be done from within a callback or action function. Sun WorkShop Visual provides simple support for the reception of a drop but not for the source of a drag.

For details on how to set up a widget as a drop site, see “Drag and Drop” on page 188.



Option: Drop site

□ Drop site

□ Animation mask <Unspecified>

□ Animation pixmap <Unspecified>

□ Pixmap depth: (0)

Import targets:

□ Animation style: (Highlight)

□ Activity: (Active)

□ Type: (Simple)

Operations: ☑ (Default) □ Copy □ Move □ Link

Drag procedure:

Drop procedure:

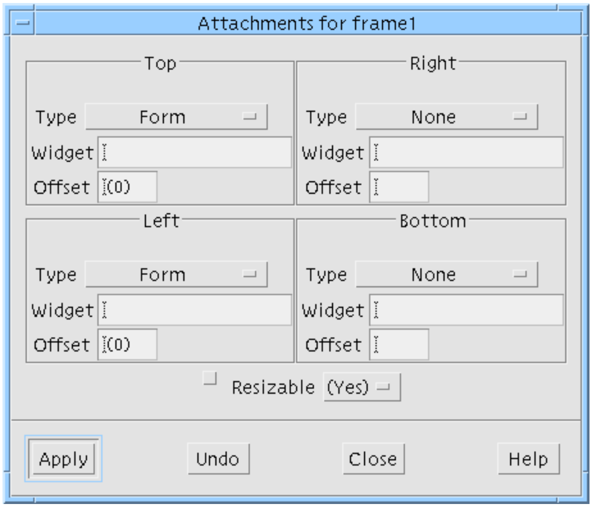
FIGURE 3-24 The Drop Site Page

Constraints Panel

Motif has two classes of widget, the PanedWindow and the Form, which are called *constraint widgets*. Widgets of these classes have a special set of resources, called *constraint resources*, that control the size and position of their children. A constraint widget maintains a separate set of constraints for each of its children. Sun WorkShop Visual lets you set them as if they were resources of the children.

To view constraint resources for any child of a constraint widget:

1. **Select a child of a contraint widget (i.e. a child of the form).**
2. **Pull down the Widget Menu and select "Constraints...".**



Attachments for frame1

Top
Type Form
Widget
Offset (0)

Right
Type None
Widget
Offset

Left
Type Form
Widget
Offset (0)

Bottom
Type None
Widget
Offset

□ Resizable (Yes)

FIGURE 3-25 Constraints Panel

Figure 3-25 shows the default constraints resource panel for the Frame child of the Form in the tutorial layout. This panel shows that the Frame's top is attached to the top of the Form and its left side is attached to the left side of the Form, with an offset of 0 pixels. This is why it is located at the upper left corner of the Form. The bottom and right side of the Frame are unconstrained.

The constraints resources which the Form imposes on its children interact in complex ways and the preferred way of setting them is by using the interactive Layout Editor, which is discussed in Chapter 4, "The Layout Editor". The constraints resource panel is mainly useful for viewing the constraints which have been set, as an adjunct to the Layout Editor. Advanced users may want to set values on the panel itself. This can be done as with any other resource panel, by typing in the new value and clicking on “Apply”.

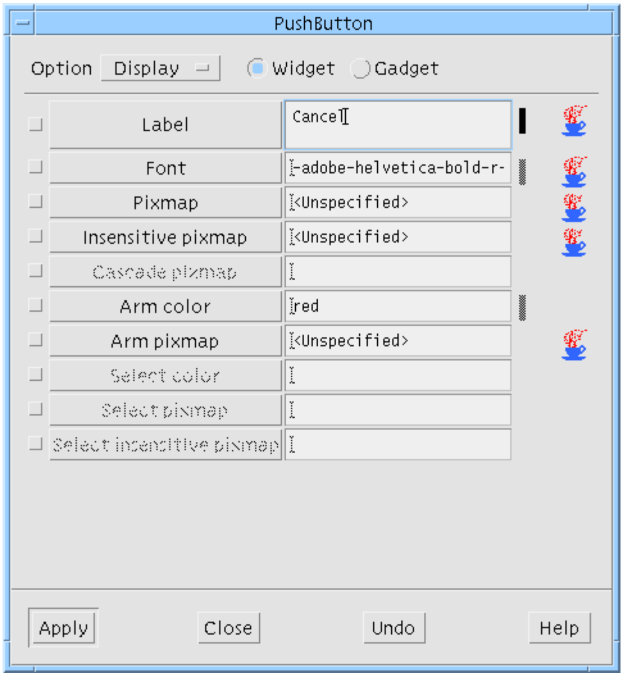
The constraints resource panel for children of the PanedWindow displays a different set of values. This panel is discussed in the section on the PanedWindow in Chapter 27, “Widget Reference”, starting on page 743.

Default Resource Settings

You may have noticed that Sun WorkShop Visual displays default resource settings in parentheses. Default settings are different from explicit settings, even if the values are the same when you build the interface. The difference is that default settings are not added to the generated code or X resource files. If your interface uses default settings, and is then run on a machine other than the one used to design it, it will use the Motif defaults for the machine on which it is running.

Many resources, such as the label on a PushButton or the number of columns in a RowColumn, are unlikely to cause portability problems. Others, such as dimensions and colors, are machine-specific. To make your interface portable, you must either use default values for such resources or put them into an X resource file at the time of code generation so they can be edited for each machine. This is discussed in “Setting up the X Resource File” on page 215.

Figure 3-26 shows a resource panel with some resources which have been explicitly set either in the current Sun WorkShop Visual session or in a previous one. The explicit settings are those marked with a change bar.



PushButton

Option Display

● Widget ○ Gadget

□	Label	Cancel
□	Font	-adobe-helvetica-bold-r-
□	Pixmap	<Unspecified>
□	Insensitive pixmap	<Unspecified>
□	Cascade pixmap	
□	Arm color	red
□	Arm pixmap	<Unspecified>
□	Select color	
□	Select pixmap	
□	Select insensitive pixmap	

FIGURE 3-26 Resource Panel with Explicitly Set Resources

To revert to the default setting for a resource for which you have entered an explicit setting:

1. **Delete all text in the resource’s text field on the resource panel.**

2. **Click on “Apply”.**

To select the default value for a resource on the “Settings” page:

3. **Select the option displayed in parentheses from that option menu.**

4. **Click on “Apply”.**

Note – The default value in this case will be the value currently being used in the dynamic display (i.e. the last value assigned to the resource). When the application is run outside of Sun WorkShop Visual the system default will be used. To see the system default in Sun WorkShop Visual, you will need to reset the widget.

The Reset Command

When you set any resource, Sun WorkShop Visual tries to apply that value to the selected widget in your dynamic display. Note that what you see in the dynamic display is a collection of widget instances. Sun WorkShop Visual does not draw pictures of widgets but actually creates them, using the same Motif function calls which your interface will use when it is running. When Sun WorkShop Visual sets a resource, it makes the appropriate Motif function call to set that resource's value for that widget.

Usually, the result of setting a value is the same as creating the widget with that value in the first place. However, this is not always the case. Sun WorkShop Visual has a command on the Widget Menu, "Reset" (<*Ctrl-T*>), which destroys the selected widget and its children and recreates them with the most recently applied resource settings. If your layout does not look or behave as expected, try using the "Reset" command. The following steps demonstrate a case where "Reset" is required:

1. **Double click on the "Help" CascadeButton widget in the hierarchy to bring up its resource panel.**
2. **Select the "Keyboard" page.**
3. **Remove your previously set mnemonic by deleting all text from the "Mnemonic" text field.**
4. **Click on "Apply".**

   Notice at this point that the "Mnemonic" text field reverts to the <*Default*> setting. However, the "Help" button in the dynamic display still has an underscore under the "H", indicating a mnemonic which is no longer present. To update the display:

5. **Select "Reset" from the Widget Menu.**

   Resetting only affects the selected widget and its children. Resetting a widget that is low in the hierarchy may leave inaccuracies elsewhere in the dynamic display. If you set many resources, it is wise to reset the Shell to guarantee that what you see is what you get.

   The "Reset" command is particularly useful when using the Form widget and its attachment resources. This topic is discussed in Chapter 4, "The Layout Editor".

6. **Reinstate the mnemonic.**

Rejected Resource Settings

Motif and other widget toolkits have rules which control legal settings of resources and, since Sun WorkShop Visual works with real instances of widgets and not simulations, any new resource setting that does not satisfy these rules is rejected. The rules include valid values for the particular widget, requirements of a parent widget such as a RowColumn and requirements of the machine you are using to build your design.

For example, if you are designing an interface for use on a large-screen workstation, you might want to set a dimension resource to a large number of pixels. If you are designing on a smaller-screen machine, you may find you cannot set the value you want even though the interface will run on the large-screen machine later. (In this situation, you could still set the width you want by using an X resource file.)

When Motif rejects a new resource setting, it does not revert to the previous setting but calculates a new value based on defaults and other resource settings in the hierarchy. This new value is reflected on the resource panel and in your dynamic display.

Resource Bindings

Previously in this chapter you have seen how to set resources for individual widgets. “Setting up the X Resource File” on page 215 describes how Sun WorkShop Visual can generate a resource file which contains all the resources you have set in the design.

The way resources and widgets are tied together is often referred to as a *binding*. One line is generated for each resource which has been explicitly set and which you have asked to be generated into a resource file. See “Resource File Syntax” on page 235 for a description of the format of resource files. “Code Generation Options” on page 218 shows how you can tell Sun WorkShop Visual which resources to generate into the resource file.

There are two methods of setting resources in Sun WorkShop Visual:

1. Setting resources on individual widgets in the resource panel

2. Using loose bindings

Which method you choose depends on the number of widgets you wish to use a resource. In large designs you may want all widgets of a particular class to look the same (all buttons to be green, for example). Loose bindings provide the means of doing this. Resources which apply to individual resources should be set using the resource panel.

Tight bindings are a way of avoiding conflicts in an application's resource file. This is explained in more detail in "Tight Bindings" on page 92.

Loose Bindings

Setting a loose binding lets you specify a default resource that will be used if no explicit resource has been set. The loose bindings dialog allows a lot of flexibility because widgets can be referred to by their widget name, their class name or even by a wildcard indicating that *any* widget fitting the description is applicable. You can also specify how *general* to make the binding. This is explained in more detail in the example below. Loose bindings are useful if, for example:

- You wish all buttons in your design to have the same background color
- You wish all buttons in a particular RowColumn to be displayed in the same font.
- You wish all "Help" or "Ok" buttons in your application to share the same label

You can achieve this by setting up loose bindings. This can be explained with a simple example.

Example Loose Binding

1. **Create a widget hierarchy containing a TopLevel Shell, a Form, another Form and three PushButtons, as shown in Figure 3-27.**

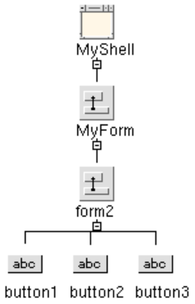


FIGURE 3-27 Hierarchy for Loose Bindings Example

2. **Name the Shell "MyShell" and one of the Forms "MyForm", as shown.**

   For resources, the widget name is the name that will be used. Setting the variable name, however, automatically sets the widget name using the same name.

3. **Select the Shell, the Form named MyForm and the first Button.**

   Select one and then hold the Shift key down while selecting the others.

4. **Select "Loose bindings" from the "Widget" menu.**

   The Loose Bindings dialog appears.

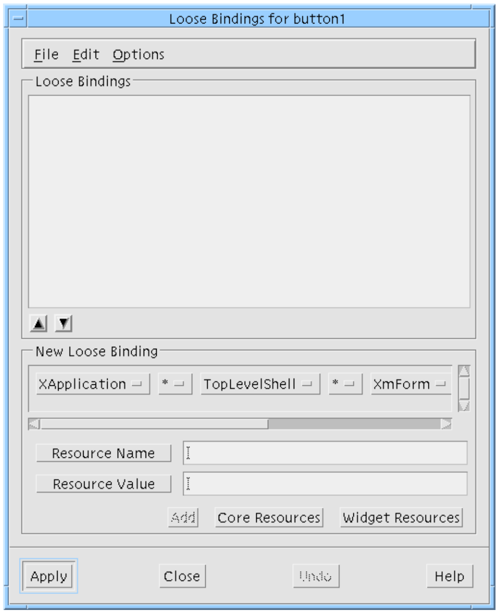


FIGURE 3-28 Loose Bindings Dialog

Loose Bindings Dialog

The Loose Bindings Dialog, shown in Figure 3-28, consists of the following areas:

Menubar

This contains three menus: "File", "Edit" and "Options". The "File" menu contains commands to Load, Merge and Inherit bindings from external resource files. These operations are described in more detail in "Resources from External Resource Files" on page 90. The "Edit" menu allows you to delete bindings as well as cut, copy and paste them. The cut and paste mechanism is important because the order of the bindings in the list is the order they are generated. The order of resources in the resource file is significant because the file is read in order by the X toolkit. If there are any conflicting resources, the later resources override previous ones. The “Options” menu contains one option “Use Inherited Bindings” which allows you to decide whether or not to use any inherited bindings which appear in your loose bindings list. This toggle is set by default.

Currently Defined Bindings

At the top of the window is the list of currently defined loose bindings. Beneath the list there is an up and a down arrow button. Use these to change the order of the bindings in the list. The order in which the bindings appear in this list is the order they are generated into the resource file.

Binding Under Construction

Underneath the list of existing bindings there is a scrolling window of option menus representing a binding for the selected widgets in the widget hierarchy.

Resource Name and Value

Beneath the representation of the binding there are two text boxes - one for the name of the resource and one for its value.

Resource Panels

At the bottom of the dialog there are buttons to bring up the core and widget resource panels. The resource panels relate to the *bottommost* widget that you have selected in the widget hierarchy (also known as the *leaf* widget). There is also a button to “Add” the binding to the list of those currently defined.

Creating the Binding

To continue with our example:

1. **Change the option menus in the Loose Bindings dialog so that the Shell and the Form are referred to by name ("MyShell" and "MyForm" respectively) and check that an asterisk separates each widget reference.**

   This shows a binding which refers to the Shell and one of the Forms by name but to the button by its Motif class name. This means that the binding refers to *all* the buttons below the named Form. In between each widget, we have selected the asterisk (*), to indicate that there can be other widgets between the named ones. The other Form is included in this way.

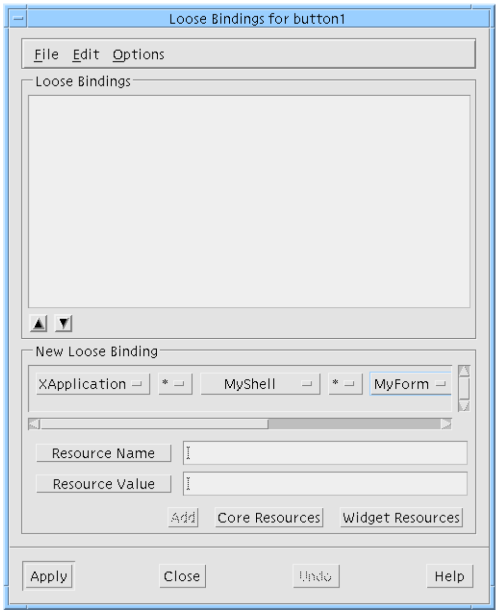


FIGURE 3-29 Widget Options for Loose Bindings

2. **Press "Widget Resources".**

   The Button widget resource panel is displayed.

3. **Set the Label resource to: `Bound`. Press "Apply" and close the resource panel.**

   The "Resource Name" text box now contains the text "labelString" which is the name for the Label resource recognized by X Windows. The "Resource value" text box contains the text "Bound" which is the value you typed into the resource panel. The following line is added to the loose bindings list:

   ```
   XApplication*MyShell*MyForm.XmPushButton.labelString: Bound
   ```

4. **Press "Apply".**

   When you next generate a resource file this will appear in it. In plain English this means:

All Push Button widgets which are children of Forms named “MyForm” which are descendants of Shells named “MyShell” (with any number of widgets in between) which are in an application of class “XApplication” will have their Label set to “Bound”.

Resources from External Resource Files

The Loose Bindings dialog “File” menu contains three items:

- Load
- Merge
- Inherit

Selecting either “Load” or “Merge” produces a File Selection Box prompting you for the name of a resource file. Selecting “Inherit Bindings” from the “File” menu displays the “Inherit Bindings” dialog.

Inherit Bindings Dialog

The Inherit Bindings dialog contains two areas: a text box and button allowing you to specify the name of the resource file from which to inherit bindings and a list of bindings from the last specified file. This is a read-only list. Pressing the button labelled “Resource File” displays a File Selection Box where you can locate a resource file.

In order to inherit the bindings displayed in this dialog, set the “Use Inherited Bindings” toggle in the “Options” menu.

When you next generate a resource file, if the toggle button in the “Options” menu is set, a reference to the named resource file is generated. This, in effect, inherits *all* the resources listed in the Inherit Bindings dialog. You can unset the “Inherited Loose Bindings” toggle if you no longer wish to inherit resources. If you unset the toggle button, no resource bindings are inherited.

Refining the Binding

The example illustrated in “Example Loose Binding” on page 86 is a simple one. You have more ways of altering the resource binding in the Loose Bindings Dialog. Each element in the binding currently being defined can be altered by means of the corresponding option menu, as shown in Figure 3-30.

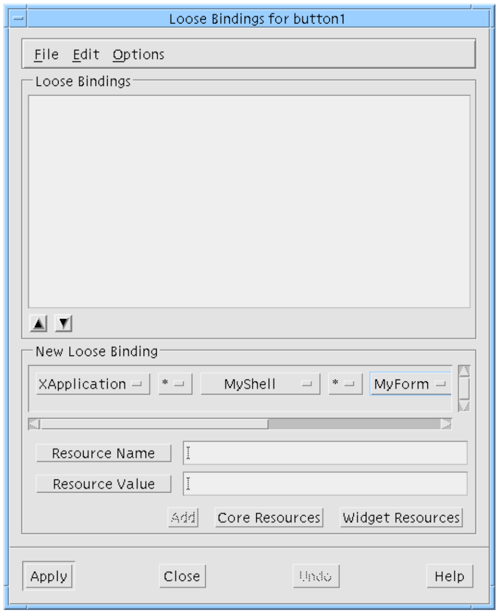


FIGURE 3-30 Binding Under Construction with Corresponding Buttons

You can refine the binding using these option menus. There are three types of refinement that can be made:

- Changing the wildcard
- Changing the reference to the widget
- Changing the resource

These are described below.

Wildcard

In between each widget name there is a wildcard character. This can be either a period (.) or an asterisk (*). Period means that the widget on the right is a direct descendant of the widget on the left. Asterisk means that there can be any number of other, unnamed widgets in between the widget on the left and the right.

Widget Reference

There are three ways of referring to widgets. You can select the Motif class name (e.g. XmPushButton for buttons, XmForm for Forms etc.), the widget name you have specified or a question mark character (?). The question mark is a wildcard meaning that there must be a widget at this point in the description but that it can be *any* widget.

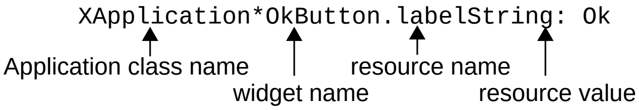
Resource

The last item in the binding currently being defined is the resource and its value. If you have set more than one resource for this binding they are listed in the corresponding option menu. You can set a loose binding for any number of resources.

Note – If the name of a widget is changed loose bindings will not be re-applied to that widget until a reset is performed.

Tight Bindings

The default resource bindings that Sun WorkShop Visual generates could lead to ambiguities if more than one widget has the same widget name within the same application class. Tight bindings are a means of naming extra widgets in a resource binding in order to lessen the possibility of ambiguity. “Resource File Syntax” on page 235 describes the structure of resource bindings as present in the resource file. The default resource binding generated by Sun WorkShop Visual for the hierarchy shown in Figure 3-31 is:



```
XApplication*OkButton.labelString: Ok
```

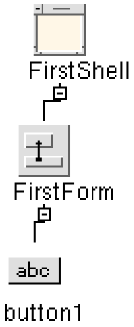


FIGURE 3-31 First Hierarchy

This example mentions only one widget explicitly, the *leaf* widget, whose widget name is "OkButton".

Note – The variable name for this widget has not been set. In resource files it is the widget name which is important.

In an application you may find that you have more than one leaf widget with the same widget name. If, however, the widgets need different resources the default resource syntax described above would not be useful since it refers to all of them. Some resource settings would then be lost. The more widgets that are named in the resource binding, the less possibility of there being a conflict over widget names in the resource file.

Example Tight Binding

Using the example above, the following shows how you can still have more than one leaf widget (in this case a PushButton) with the same name but with different labels.

1. **Create the hierarchy shown in Figure 3-31, making sure that you have used the names shown for the widget names.**

2. **Select another Shell (of any type) from the palette and create a second hierarchy as shown in Figure 3-32. Give the button the widget name "OkButton", the same widget name as the button in the first hierarchy.**

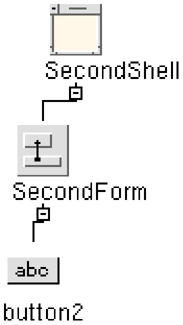


FIGURE 3-32 Second Hierarchy

3. **Specify the label: `Ok` for the button in the first hierarchy**
4. **Specify the label: `Apply` for the button in the second hierarchy**
5. **Select the widget FirstShell in the first hierarchy and bring up its core resource panel**
6. **Select the "Code Generation" page**
7. **Set the toggle labelled "Include in Resource Bindings"**

   This is the toggle which makes a widget's resource bindings "tight".

8. **Press "Apply"**

   When the resource file is next generated, the widget with the tight resource binding will appear as follows:

   ```
   XApplication*FirstShell*OkButton.labelString: Ok
   ```

9. **Now do the same for the second hierarchy, including the widget SecondShell in the resource binding.**

   The binding for the second hierarchy looks like this:

   ```
   XApplication*SecondShell*OkButton.labelString: Apply
   ```

   Resources for the two buttons with the same widget name are now easily distinguished. You can add any number of widgets into the resource binding (in this example we could have added the Form as well). This would make the binding *tighter* and less open to ambiguity.

Tight Resource Binding Recommendation

It is recommended that you set the “Include in Resource Bindings” toggle on all Shells in your design. This will not cause any unwanted effects and will prevent most ambiguities.

Comparison of Resource Generation

If you explicitly set the Label resource of the first button in the loose bindings example shown in Figure 3-27, but you do not set a loose binding, the following line is generated into the resource file:

```
XApplication*button1.labelString:Bound
```

If you have used tight bindings, which are described in “Tight Bindings” on page 92, to set a tight binding for the Shell, MyShell, without specifying any loose bindings, the following line is generated into the resource file:

```
XApplication*MyShell*button1.labelString:Bound
```

If you have set a loose binding to encompass all the buttons, as described in “Example Loose Binding” on page 86, the following line is generated into the resource file:

```
XApplication*MyShell*MyForm*XmPushButton.labelString:Bound
```

Where to Look for More Information on Resources

For more information on widget resources see Chapter 27, “Widget Reference”, starting on page 743. This chapter contains a summary of the most commonly used resources for each of the Motif widget classes. While this summary is necessarily brief, it will help you get started.

There are many books available that provide a more complete discussion of Motif widget resources. The *Motif Programming Manual* includes a summary which is both thorough and readable. Several other useful books are listed in Appendix E, “Further Reading”.

If you are using Sun WorkShop Visual with additional widgets, you should also consult the documentation provided by your widget developer.

CHAPTER 4

The Layout Editor

Introduction

The next step in designing your interface is to rearrange the widgets geometrically. The DialogTemplate automatically places the menu bar at the top of the window and arranges the buttons at the bottom. However, the arrangement of widgets inside the Form is up to you.

Figure 4-1 shows the present appearance of the tutorial interface and how it will appear when you finish making the modifications in this chapter.

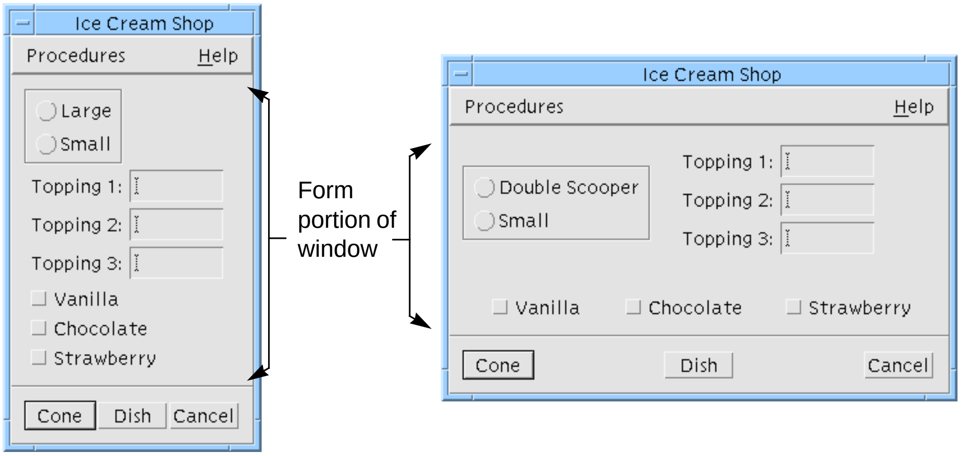


FIGURE 4-1 Default and Modified Layout for Tutorial Interface

The arrangement of widgets inside the Form is called the *layout*. To make these changes in the layout, you will perform the following steps with Sun WorkShop Visual's interactive Layout Editor:

1. **Move widgets around to the approximate layout you want.**
2. **Attach the top and left side of the Frame to the sides of the Form at a fixed offset.**
3. **Attach the RowColumn to the Frame at a fixed offset.**
4. **Align the tops of the three ToggleButtons located at the bottom of the layout.**
5. **Set position attachments for proportional spacing of the three ToggleButtons.**

Concepts

Several classes of Motif widgets can impose geometric rules on their children. You have already seen that the children of a RadioBox, RowColumn and MenuBar are laid out in specific ways. In a MenuBar, the CascadeButtons are laid out in a single row. In a RowColumn, all children are laid out in a grid.

Three types of widget, Form, BulletinBoard and DrawingArea, allow more flexible layout of their children. These three widget classes are called *layout widgets*. Sun WorkShop Visual provides an interactive Layout Editor for laying out the children of these widgets.

Attachments

Attachments are constraints that force a widget to be in a certain location relative to the layout widget or to another child of the layout widget. All three types of layout widget let you attach their children's upper left and lower right corners at any *x,y* location relative to the layout widget. If you constrain the upper left corner of a widget, you fix its location. If you also constrain its lower right corner, you also fix its size. These are the only attachments offered by the BulletinBoard and DrawingArea widgets.

Form Attachments

The Form widget lets you attach a side of a child widget to a side of the Form at a specified distance (measured in pixels). This has the effect of positioning the widget at a specific *x,y* location, like the attachments provided by the BulletinBoard and DrawingArea. In the case of the Form, this type of attachment is called a *Form attachment* to distinguish it from other types of attachments available for the Form.

Position Attachments

The Form also has *position attachments*, which are specified as a percentage of the total width or height of the Form. When you use position attachments, widgets get farther apart when the window gets larger so that the interface can take advantage of extra space when it is available.

Widget Attachments

Widget attachments let you attach two of the Form’s children to each other at a specified absolute distance (measured in pixels). Attachments can be made edge to edge or top to bottom. Facilities are also provided for you to align and distribute a group of widgets.

These additional features of the Form allow you to design a layout which retains its appearance when the main window or an individual widget is resized. For example, by attaching a widget to both sides of the Form, you can make it stretch when the main window becomes larger. By attaching two widgets edge-to-edge, you can ensure that the spacing between them will be preserved even if one of them moves or changes size.

Displaying the Layout Editor

You can display the Layout Editor for any layout widget in your hierarchy by selecting it and either selecting “Layout” from the Widget menu or clicking on the Layout button on the toolbar, as shown in Figure 4-2.

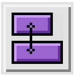


FIGURE 4-2 Layout Toolbar Button

To edit the layout in your Form:

1. **Select the Form.**
2. **Click on the Layout button on the toolbar.**

   This displays the dialog shown in Figure 4-3.

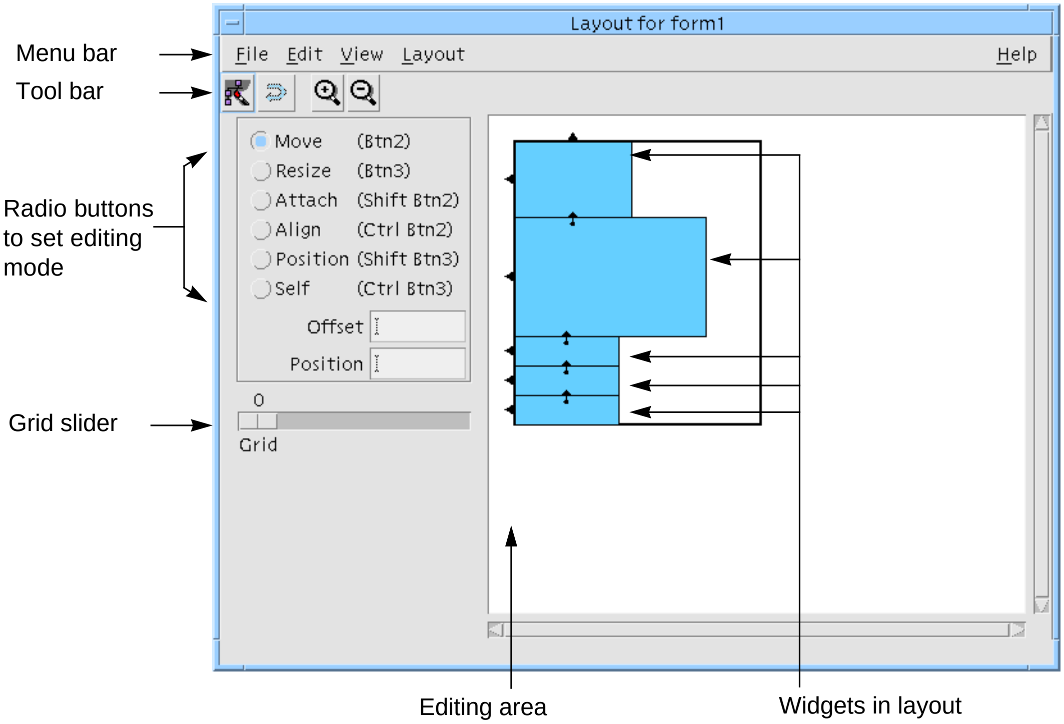


FIGURE 4-3 Layout Editor Dialog

Editing Area

The *editing area* displays a sketch of the layout. Widgets in the editing area are displayed schematically as boxes within a larger box which represents the Form. In Figure 4-3, the boxes represent, from top to bottom, the Frame, the RowColumn and the three ToggleButtons.

If widgets overlap, the outlines are still visible so that you can see where they are. When you select a widget, the smallest enclosing widget is selected enabling you to select a widget which may be obscured by a larger one.

The Layout Editor shows the Form’s children but nothing lower on the hierarchy. For example, you cannot see the RadioBox and its ToggleButtons inside the Frame nor the Labels and TextFields inside the RowColumn. You also cannot see the MenuBar or the PushButtons, which are outside the Form.

Note – The Layout Editor has its own menu bar, toolbar and several command buttons. Some of the Layout Editor commands have keyboard accelerators. If you use them, be sure that the Layout Editor screen has the input focus as some of the same characters are also used as accelerators within the main Sun WorkShop Visual window where they have different functions.

Editing Modes

A set of radio buttons on the left side of the editor screen lets you select one of several *editing modes*. Selecting an editing mode assigns that function to mouse button 1.

You can also use any mode, regardless of which is currently selected, by using the mouse button sequence indicated next to that radio button. For example, you can always use *mouse button 2* to move a widget, or <*Shift-button 2*> to set an attachment.

There are six modes:

- Move. See “Rough Layout: the Move Mode” on page 107.
- Resize. See “Resize Mode” on page 130.
- Attach. See “Attachments Between Widgets” on page 113.
- Align. See “Aligning Widgets: the Align Mode” on page 118. This is slightly different from using the align toolbar buttons as described in “Aligning Widgets: Group Alignment” on page 120.
- Position. See “Proportional Spacing: the Position Mode” on page 126.
- Self. See “Self Mode” on page 128.

Selection: Single, Primary, and Secondary

To select a widget, click over it in the editing area. To select additional widgets, hold down the Shift key while selecting them. The primary selection is always displayed with a thick border, secondary selections are displayed with a dotted border. The last widget to be added to the selection is always the primary selection. You need to know which is the primary selection because alignments are performed by aligning *to* the primary selection.

Layout Editor Toolbar

Below the menu bar is a toolbar containing the following buttons which are also available from the menus:

See “Reset” on page 103.

See “Undo” on page 103.

See “Zoom In, Zoom Out” on page 104.

See “Zoom In, Zoom Out” on page 104.

Layout Editor File Menu

The File menu contains one item, “Close”, which removes the Layout Editor window from the screen.

Layout Editor Edit Menu

The Edit menu provides three functions.

Undo

“Undo” reverses the last operation done in the Layout Editor. Selecting “Undo” repeatedly lets you step back through multiple operations. As with other Layout Editor commands, you may need to “Reset” before you can see the effect of “Undo”.

Reset

“Reset” (*<Ctrl>-T*) destroys the current instance of the layout widget, along with its children, and re-creates your dynamic display.

Sometimes new constraints are not reflected accurately in the dynamic display window until after you reset the layout widget. Therefore you should reset often when using the Layout Editor, especially if a change does not produce the result you expect.

If container widgets within a Form do not properly display after a reset, select a widget higher up the hierarchy and do another reset.

Layout Editor View Menu

The View Menu of the Layout Editor provides four useful display commands.

Edge Highlights

This option highlights the widget edge closest to the pointer when the pointer is inside the widget. Any attachment you make is applied to the highlighted edge of the widget. “Edge Highlights” is *on* by default.

- **Move the pointer around the sketch of the Form and note that, whenever the pointer is inside the box representing a widget, the edge nearest the pointer is highlighted.**

Annotation

This option displays an identifying string inside each box in the editing area. When you pull down the View Menu and select “Annotation”, a pullright menu appears with a choice of “Widget names” or “Class names”. “Widget names” (<*Ctrl*-*W*>) displays the variable name of each widget. “Class names” (<*Ctrl*-*N*>) displays the class of each widget, such as “Frame” or “RowColumn”. Selecting one option disables the other. Selecting the same option when it is set removes all annotation from the display.

The annotation options are illustrated in Figure 4-4.

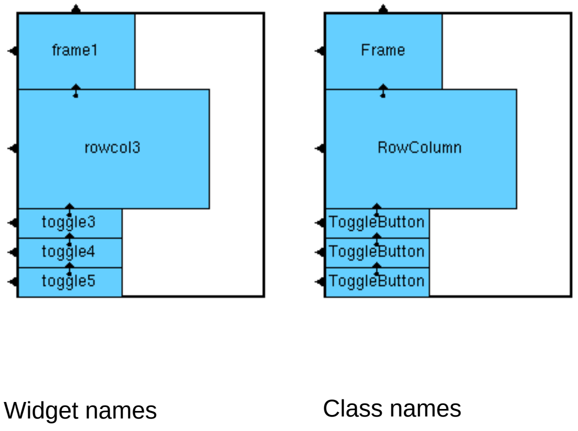


FIGURE 4-4 Annotation Options

- **Pull down the View Menu and select “Class names” from the “Annotation” submenu.**

Zoom In, Zoom Out

Use “Zoom In” to increase the display scale and “Zoom Out” to decrease the scale.

Layout Editor Layout Menu

The Layout Menu provides some layout commands in addition to those available in the editing modes palette which is described in “Editing Modes” on page 101. There are two functions available from this menu - align and distribute.

Align

Pressing “Align” displays a pull-right menu containing the various types of alignment available. These functions are only enabled if there is more than one widget selected. See “Aligning Widgets: Group Alignment” on page 120 for a detailed description of the alignment functions available from this menu item.

Distribute

Pressing “Distribute” displays a pull-right menu containing two further pull-right menus which lead to the distribution options available. These functions are only enabled if there are more than two widgets selected. See “Distribute” on page 123 for a detailed description of the distribution functions available from this menu item.

Grid

To display a grid on your layout, use the grid slider, situated in the bottom left corner of the dialog, to select a spacing from 2 to 50 pixels. The number of pixels is considered to be at 1:1 scale so that the grid will scale with the layout if you use the Zoom commands.

The “Move” and “Resize” modes snap to the grid if one is visible. When you move a widget, its top left corner snaps to the nearest grid intersection. Likewise, when you resize a widget, its lower right corner snaps to the grid.

You can disable the grid by setting the slider to zero.

- **Set the grid slider to 10 pixels.**

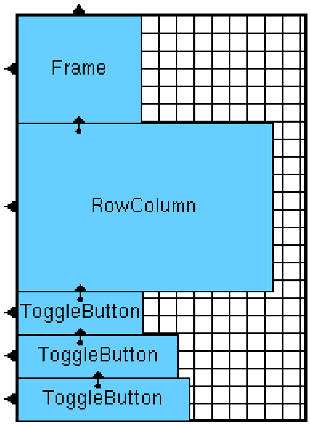


FIGURE 4-5 Tutorial Layout with 10 Pixel Grid

Resize Policy

Most Form resources should be left at their default values. You could, however, consider resetting the “Resize Policy” resource to “Grow”.

1. **Double click on the Form widget or press the “Resources” button in the toolbar to display the Form’s resource panel.**
2. **Select the “Settings” page of the resource panel.**
3. **Set “Resize policy” to “Grow”.**
4. **Click on “Apply” and then “Close”.**

If you do not set this resource, the Form always shrinks immediately to the minimum size when you move widgets around, which can be annoying. With a policy of “Grow”, although you may sometimes have extra blank space in the layout, it goes away when you reset the Form.

Understanding the Default Layout

The attachments are displayed in the Layout Editor using symbols as shown in Figure 4-6.

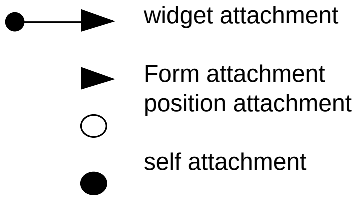


FIGURE 4-6 Symbols used in the Layout Editor

Your layout already has two kinds of attachments: Form attachments and widget attachments. Form attachments are shown as arrowheads on one edge of the widget. The Frame, which is the top widget in the default layout, has two such attachments - to the top and the left side of the Form. The other four widgets are each attached to the left side of the Form. All the widgets are stacked top to bottom in the order in which they were added to the hierarchy.

Each of the four lower widgets is attached to the widget above it. A widget attachment is shown as an arrow with an arrowhead at one end and a small filled circle at the other.

Default MenuBar Attachment

In addition to the above default attachments, a MenuBar is treated as special and given a default attachment from its right edge to the right side of the Form. This is to make a MenuBar fill the top of the window and resize appropriately.

Rough Layout: the Move Mode

The “Move” command lets you drag a widget to a new location. This sets two Form attachments which fix the widget’s upper left corner at that point. The best way to start arranging any layout is to use this mode to place all the widgets approximately where you want them; you can then use the other modes to specify widget positioning and resize behavior more precisely.

Removing Attachments on Move

When you move a widget, Sun WorkShop Visual removes all attachments *from* that widget. Attachments *to* the widget are preserved. You can prevent this from happening by pressing the Control key while moving a widget. Sun WorkShop Visual removes attachments by default because other functions such as align and distribute rely on making attachments. Often the new attachments conflict with the old and cause the Form to report errors. Once into this state it is often difficult (though not impossible) to recover - see “Circular Attachments” on page 116. You can avoid this situation by allowing Sun WorkShop Visual to remove all attachments before performing one of the layout functions.

Using Move

To invoke the “Move” mode:

1. **Click on the “Move” toggle.**

   Once in “Move” mode, you can drag widgets around in the layout with mouse button 1. You can also move widgets using mouse button 2, regardless of the current mode.

2. **Place the pointer inside the box corresponding to the RowColumn.**

3. **Hold mouse button 1 down and drag the RowColumn to the right until it overlaps the right edge of the Form.**

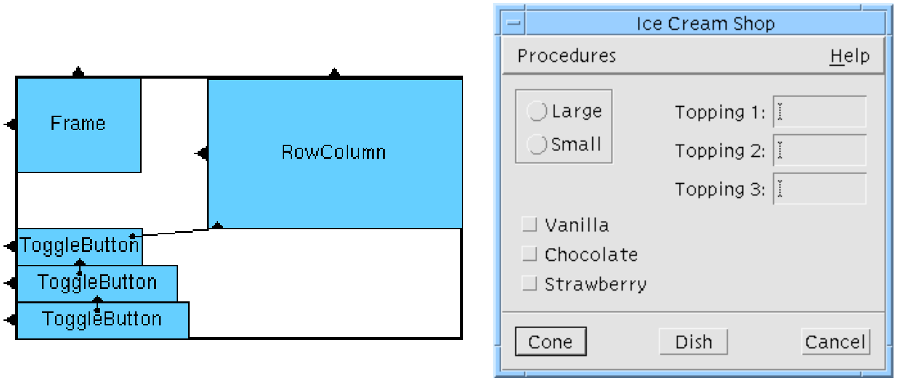
   To do this, you have to drag the RowColumn so that its right margin extends beyond the edge of the Form. This is acceptable. When you release the mouse button, the Form automatically resizes to allow room for all its children.

4. **Drag the RowColumn up until its top is aligned with the top of the Frame.**

5. **Reset the Form.**

   If the display is not correct, reset the Shell, select the Form and then re-display the layout editor.

   Figure 4-7 shows what your layout looks like now and the resulting dynamic display. The DialogTemplate automatically adjusts the size of the MenuBar and the spacing of the PushButtons to accommodate the new width of the Form.



Layout

Dynamic display

FIGURE 4-7 Rough Layout as Shown in the Layout Editor

How “Move” Mode Works

The tops and left edges of the RowColumn and Frame have arrowheads, indicating that they are attached to the top and left side of the Form. “Move” works by setting two attachments to the Form to fix the upper left corner of the widget at its new location. You can make these attachments either at a distance of zero (so that the widget touches the side of the Form) or at a non-zero distance to produce a space between the widget and the Form. This distance is called the *offset*. In this case, the left edge of the RowColumn is attached to the Form at a non-zero offset. Offsets are discussed in a later section of this chapter. In the “Move” mode, Sun WorkShop Visual calculates the offset based on the location of the pointer.

One Attachment Replaces Another

There can be only one attachment on each side of a widget. “Move” breaks any attachments that are already on those edges and then sets two new attachments on the top and left edges of the moved widget. In the default layout, the top of the RowColumn was attached to the bottom of the Frame. This attachment no longer exists because “Move” has set a new attachment on the top of the RowColumn. “Move” breaks the attachments made from a widget but preserves any attachments made to it from other widgets.

Note – You can retain attachments during a “Move” by holding down the Control key as you move the widget.

One Attachment Affects Another

Note that when you moved the RowColumn widget up, the three ToggleButtons moved with it. This happens because of the attachments between the widgets. Because the top of the first ToggleButton is attached to the bottom of the RowColumn, when you move the RowColumn up, the top ToggleButton also has to move. The other two ToggleButtons also move up in a chain reaction since the second is attached to the first and the third to the second.

The attachment from the ToggleButton to the RowColumn was not removed when you moved the RowColumn. This is because this attachment belongs to the ToggleButton and not to the RowColumn. This distinction is discussed later in this chapter.

Offsets

The tops of the RowColumn and the Frame are now lined up. However, there is an important difference in the way they are attached to the Form. The top of the Frame has a default attachment which was left over from the default layout. The RowColumn has a zero attachment which was set when you used the “Move” mode in Step 1 on page 108.

Default vs. Explicit Offsets

Default offsets are controlled by the “Horizontal spacing” and “Vertical spacing” resources of the Form. Explicit offsets override Default offsets through actions such as Move or Align in the Layout Editor. The Frame is 0 pixels from the top of the Form because the spacing resources are both 0 by default.

To see the effect of resetting the spacing:

1. **Select the form and click on the “Resources” button on the toolbar.**

   On the Form resource panel:

2. **Select the “Display” page.**
3. **Double-click in the “Horizontal spacing” box and type:** **`20`**
4. **Double-click in the “Vertical spacing” box and type:** **`20`**
5. **Click on “Apply” and then “Close”.**

In the Layout Editor:

6. Click on “Reset”.

Figure 4-8 shows the results. All attachments with default offsets, including those on the Frame and the three ToggleButtons, now use the 20 pixel spacing. The RowColumn doesn’t move because its attachments were set with “Move” and have explicit offsets that do not refer to the spacing resources. The result is that the Frame and RowColumn are no longer aligned.

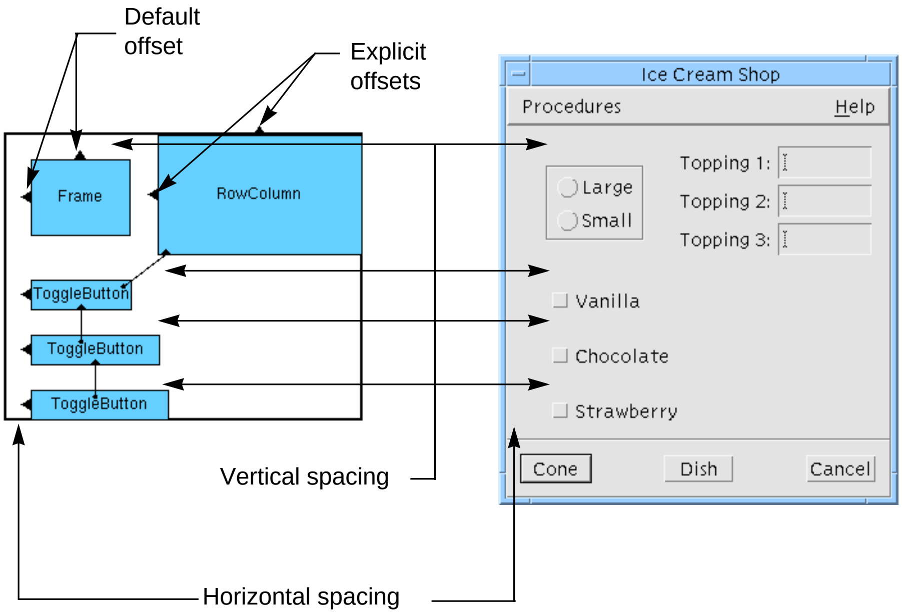


FIGURE 4-8 Effect of Resetting Vertical and Horizontal Spacing

The additional spacing between the ToggleButtons forces the entire Form to become larger. The DialogTemplate also resizes to accommodate the Form.

An advantage of default offsets is that they let the user control the amount of spread in the layout at run time. The main disadvantage of default offsets is that they require all spacings in your layout to be the same, which may not be what you want. Also, you should be careful not to confuse default and explicit offsets, since default offsets may change while explicit ones remain the same. For example, your Frame and RowColumn lost their alignment when you changed the spacing because one has an explicit offset and the other a default offset from the top of the Form. The Layout Editor screen does not distinguish between explicit and default offsets.

Attachments to the Form

You can attach a widget to the Form by dragging from just inside the widget's edge to a point just outside the side of the Form. This can be done with button 1 in "Attach" mode or with <*Shift-button 2*> in any mode. Attachments are set using the offset value in the "Offset" field. If the "Offset" field is empty, a default offset is used.

1. **Click on the "Attach" toggle.**

   Note that when you are in the "Attach" mode or have <*Shift-button 2*> down, the pointer becomes a set of crosshairs.

   Now replace some of the default attachments with attachments that use explicit offsets of 0 pixels.

2. **Click in the "Offset" box and type:** `0`

   Attach the left edge of the Frame to the left side of the Form:

3. **Place the crosshairs just inside the left edge of the Frame so that the left edge highlights.**

4. **Hold down mouse button 1 and drag to a position just outside the left side of the Form.**

5. **Release the mouse button.**

   The new attachment, like the old one, appears as a filled triangle on the side of the Frame. You can see its effect because the explicit 0 offset moves the Frame over to the side of the Form. If this does not happen, try setting the attachment again.

   Do the same thing for the "Vanilla" ToggleButton:

6. **Place the crosshairs just inside the left edge of the top ToggleButton so that the edge highlights.**

7. **Drag with mouse button 1 to a position just outside the left side of the Form.**

8. **Release the mouse button.**

   The ToggleButton moves over to the side of the Form.

   The top of the Frame looks good at its present location. The 20 pixel offset from the top of the Form centers the Frame with respect to the RowColumn. However, this should be an explicit offset, not a default, so that it will remain constant if the Form spacing resources change. To change to an explicit offset, you must replace the attachment.

9. **Double-click in the “Offset” field and type: `20`**

10. **Place the crosshairs just inside the top of the Frame so that the edge highlights.**

11. **Drag with mouse button 1 to a position just outside the top of the Form.**

12. **Release the mouse button.**

    Figure 4-9 shows the result. If your dynamic display does not look the same

13. **Click on “Reset”.**

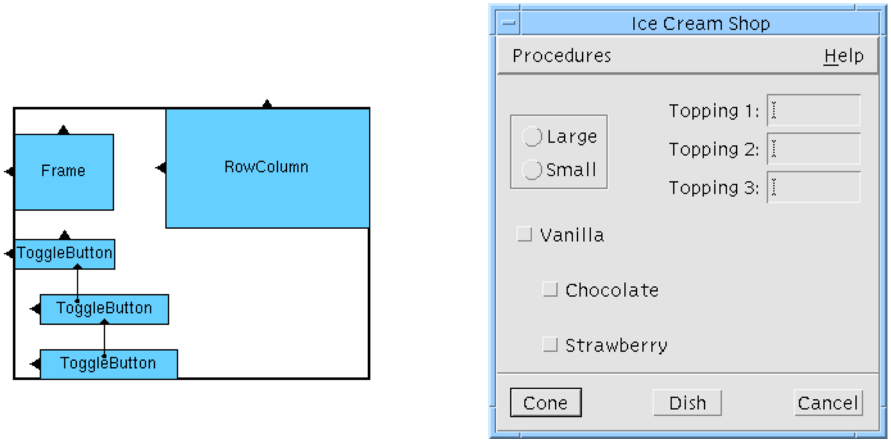


FIGURE 4-9 Form Layout with 20 Pixel Form Spacing

Attachments Between Widgets

Setting a widget attachment between two widgets in the Form is similar to attaching a widget to one side of the Form. To attach a pair of widgets, you simply drag the crosshairs from just inside one of the widgets to just inside the other.

Attaching Widgets Edge to Edge

To attach two widgets edge to edge, either touching or with an offset, you either attach the right edge of one to the left edge of the other, or the top of one to the bottom of the other.

You should still be in “Attach” mode from the last section. Attach the left side of the RowColumn to the right side of the Frame:

1. **Double-click in the “Offset” box and type: `50`**

2. **Position the pointer just inside the RowColumn’s left edge so that the left edge highlights.**

3. **Hold down mouse button 1 and drag to a point just inside the Frame, until the right edge of the Frame highlights.**

4. **Release button 1.**

   The new attachment appears as an arrow with an arrowhead pointing at the right edge of the of the Frame and a filled circle on the center of the left edge of the RowColumn. The RowColumn repositions itself with a gap of 50 pixels from the right edge of the Frame.

   Although this does not change your layout much now, there is a significant advantage to this kind of attachment if the strings inside the Frame are likely to change. The RowColumn is now positioned relative to the right side of the Frame and not relative to the side of the Form. Even if the Frame grows, the RowColumn preserves the 50 pixel distance.

5. **Double-click on the first radio button in the radio box in the construction area.**

6. **Go to the “Display” page of the resource panel and change the button’s label to: `Double Scooper`.**

   The results are shown in Figure 4-10.

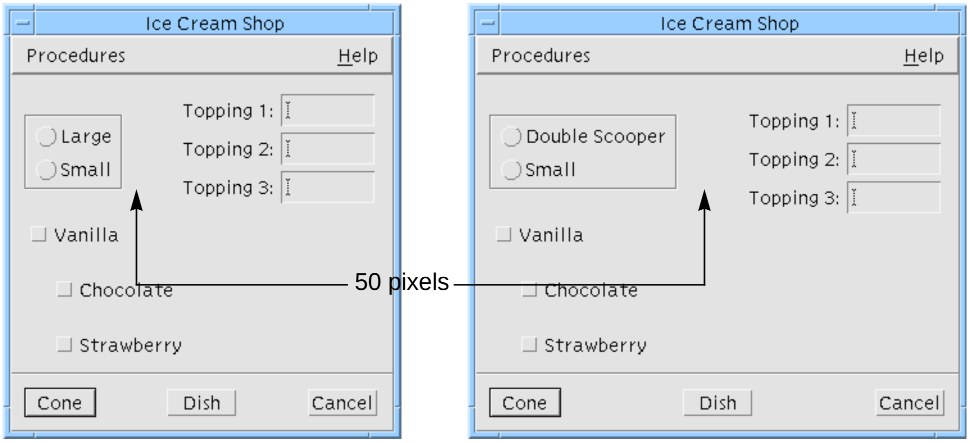


FIGURE 4-10 Frame Resize Behavior

Direction of Attachment

Attachments are not symmetrical. When you create an attachment by dragging the pointer from Widget A to Widget B, the attachment is said to *originate* from Widget A. To indicate this, Sun WorkShop Visual draws an arrow from inside Widget A to Widget B. The attachment you have just made originates from the RowColumn.

Attachments apply only to the widget from which they originated. For example, the attachment you have just made constrains the RowColumn to a certain position relative to the Frame. If the Frame is moved or resized, the RowColumn moves in turn. If the RowColumn is moved or resized, the Frame is unaffected.

Attachments Affect Only One Coordinate

Note that the top ToggleButton has an attachment to the bottom of the RowColumn. This attachment is left over from the default layout and has the default offset. It is therefore controlled by the Form's spacing resources, which are still set at 20 pixels.

You were able to move the RowColumn away from the first ToggleButton because of the following rules:

1. Attachments on the top or bottom of a widget only affect its *y* coordinate.
2. Attachments on the left or right edge of a widget only affect its *x* coordinate.

The top of the first ToggleButton is still 20 pixels (the current vertical spacing) from the bottom of the RowColumn. Because there is no attachment between the ToggleButton's left or right side and the RowColumn, the RowColumn's position in the horizontal dimension has no effect on the ToggleButton.

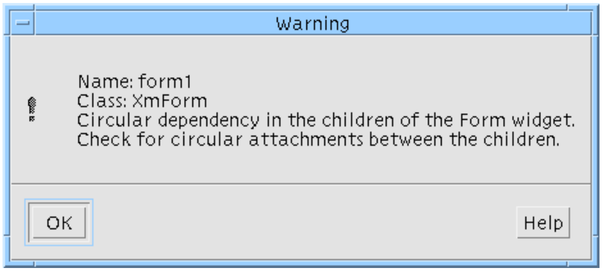
Change the spacing of this attachment to 10 pixels:

1. **If it is not already selected, click on the Form in the construction area.**
2. **Double-click in the "Offset" field and type: `10`**
3. **Position the crosshairs just inside the top edge of the top ToggleButton so that the edge highlights.**
4. **Hold mouse button 1 down and drag to a position just inside the bottom edge of the RowColumn so that the edge highlights.**
5. **Release mouse button 1.**

The new attachment replaces the old one and the top ToggleButton adjusts its position.

Circular Attachments

The Motif rules for the Form widget prohibit you from attaching Widget A to Widget B and then also attaching Widget B to Widget A. This is called a *circular attachment*. Any larger attachment loop, such as attaching Widget A to B, B to C and C to A, is also considered circular and results in an error. If your layout contains such a loop, Sun WorkShop Visual displays the warning message shown in Figure 4-11 and you must break one or more attachments to eliminate the loop.



Name: form1
Class: XmForm
Circular dependency in the children of the Form widget.
Check for circular attachments between the children.

FIGURE 4-11 Circularity Warning Message

Circularity is only a problem with widget attachments. Attachments to the Form and position attachments (see “Proportional Spacing: the Position Mode” on page 126) cannot produce a circular attachment because these attachments can only originate from the child widget and not from the parent Form.

Method for Avoiding Circularity

A good method for avoiding circularity is to make all attachments between widgets point in only two directions, usually up and left. When you lay out your interface, start at the upper left corner and work down and to the right. Whenever you attach two widgets, make the attachment originate from the widget that is below or to the right. In this way, all the attachment arrows point the same way and you avoid accidental circular attachments.

Removing Attachments

Use any of the following methods to remove attachments in the Layout Editor:

1. Set a new attachment of any type on that edge of the widget. This removes and replaces the old attachment.

2. Use “Move” to reposition the widget. This removes all attachments that originated from it.

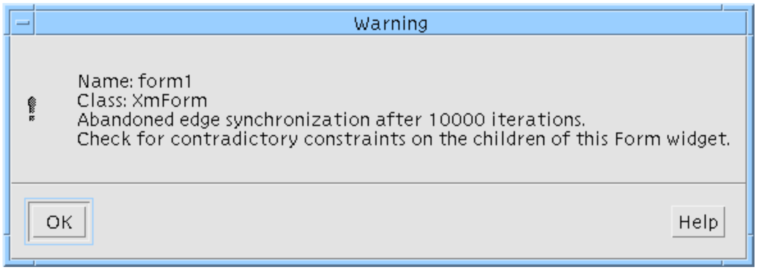
3. Using the “Attach” mode (<*Shift-Button 2*>), click just inside the widget on the edge where the attachment originates. This removes the attachment without setting a new one. The “Edge Highlights” mode can help you position the crosshairs properly.

4. Click on “Undo” to remove the last attachment added.

Motif requires that each widget have at least two edges attached. If you remove all attachments, the Form supplies simple Move-type attachments, based on the widget’s last location, when you reset.

Contradictory Attachments

You can specify attachments that contradict one another without being circular. For example, you might attach the left edge of a widget to the right side of the form using a positive offset. When you reset a Form that has contradictory attachments, Motif tries to calculate a layout that will satisfy all of them. If a satisfactory layout has not been found after a large number of iterations of a loop, the loop is broken and Sun WorkShop Visual displays the warning message shown in Figure 4-12. In these circumstances, some widgets may appear very small, or the Form itself may be resized very wide or long, until you remove the attachment that is causing the problem.



Name: form1
Class: XmForm
Abandoned edge synchronization after 10000 iterations.
Check for contradictory constraints on the children of this Form widget.

FIGURE 4-12 Bailed out Warning Message

As with circular attachments, contradictory attachments must be removed before you can proceed.

Limit on Number of Attachments

A widget can have only one attachment originating from each of its four edges. That attachment can be of any type: a Form attachment, a widget attachment or a position attachment (see “Proportional Spacing: the Position Mode” on page 126). Whenever you specify a new attachment originating from one edge of a widget, it replaces any attachment that was already there.

There is no limit to the number of attachments *to* a widget, provided they all originate from other widgets.

Aligning Widgets: the Align Mode

You can align the tops of two widgets by attaching them top to top with an offset of zero. An easy way to do this is to select the “Align” mode of the Layout Editor. The “Align” mode is simply an attachment with an explicit zero offset. You can also align a pair of widgets on any other edge: bottom to bottom, left to left or right to right. Be careful to avoid circularity when you use this feature.

To align the tops of the first two ToggleButtons:

1. **Use “Move” (button 2) to move the bottom two ToggleButtons into a rough horizontal row.**

   Do not move the top ToggleButton.

   So that you can see the effects of “Align” and, later, “Distribute”, deliberately leave the tops of the widgets and the spacing of their edges, slightly uneven, as shown in Figure 4-13.

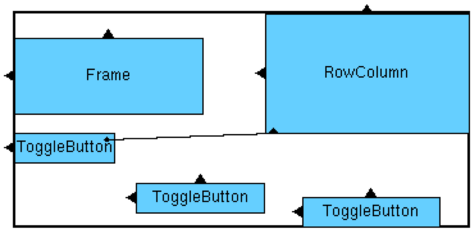


FIGURE 4-13 ToggleButtons Before Alignment

2. **Click on the “Align” toggle.**
3. **Position the crosshairs just inside the top of the middle ToggleButton.**

4. **Hold mouse button 1 down and drag to a position just inside the top of the first (left) ToggleButton.**
5. **Release the mouse button.**

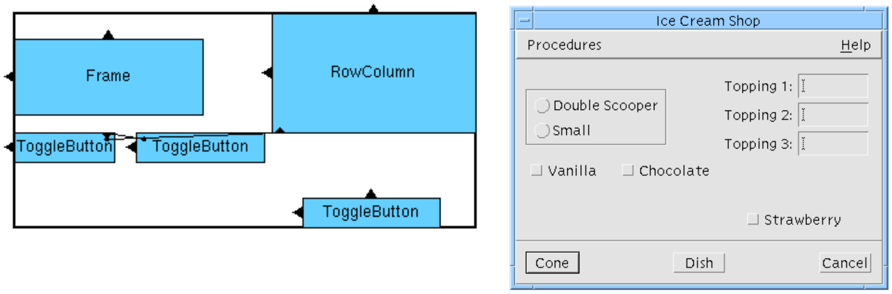


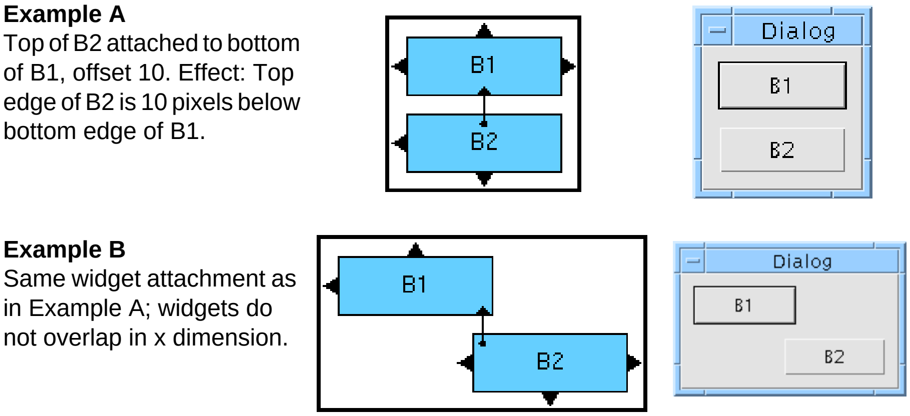
FIGURE 4-14 After Aligning First Two ToggleButtons

How Alignment Works

To understand how this works, remember that an attachment affects a widget's position in only one dimension. For example, if you attach the top of Widget 1 to the bottom of Widget 2:

```
top_1 = bottom_2 + offset
```

where *top_1* represents the *y* coordinate of the top of Widget 1 and *bottom_2* the *y* coordinate of the bottom of Widget 2. This is true whether or not the two widgets overlap in the horizontal dimension.



Example A
Top of B2 attached to bottom of B1, offset 10. Effect: Top edge of B2 is 10 pixels below bottom edge of B1.

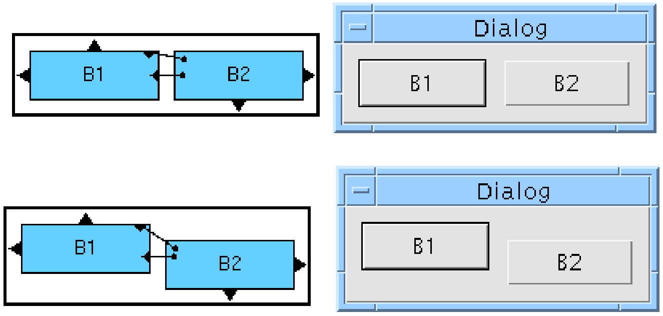
Example B
Same widget attachment as in Example A; widgets do not overlap in x dimension.

FIGURE 4-15 Attachment of Two Widgets Top-to-Bottom

Similarly, if you attach two widgets top to top:

```
top_1 = top_2 + offset
```

If the offset is 0, the tops of the two widgets have the same *y* coordinate; that is, they are aligned.



Example A
Top of B2 attached to top of B1, offset 0. Effect: Tops of widgets are aligned.

Example B
Top of B2 is attached to top of B1, offset 10. Effect: Top edge of B2 is positioned 10 pixels below top of B1.

FIGURE 4-16 Attachment of Two Widgets Top-to-Top

Example A of Figure 4-16 shows the type of attachment used by the “Align” mode. Example B shows a similar attachment with a non-zero offset. The effect is similar to an alignment but with a step effect.

Aligning Widgets: Group Alignment

You can align widgets in pairs using either the “Align” mode or by using the “Attach” mode with a 0 offset to attach their left, right, top or bottom edges. The align buttons in the Layout Menu provide a quick way to align a group of widgets. The attachments set by these are the same kind used to align widgets in the “Attach” or “Align” mode.

These buttons are only enabled if more than one widget is selected. For more details on selecting more than one widget, in particular specifying the primary selection, see “Selection: Single, Primary, and Secondary” on page 102.

The Align Functions

There are six align functions on the pull-right menu available from “Align” in the Layout Menu. These are:

- Align left edges.
- Align left and right edges.
- Align right edges.
- Align top edges.
- Align top and bottom edges.
- Align bottom edges.

Using the Align Functions

To see the effects of group alignment:

1. **Use “Move” mode or mouse button 2 to move the second ToggleButton back out of alignment.**

2. **Use “Move” mode or mouse button 2 to move the right ToggleButton so that its right edge is aligned with the right edge of the Frame.**

Now align the tops of the three ToggleButtons as a group:

3. **On the Layout Editor screen, click on the right ToggleButton.**

In “Move” mode, widgets are drawn with a thicker border as you click on them to indicate that they are selected. After you click on the first widget, you must use *<Shift-Button 1>* to add widgets to the selection group. The *primary* selection is shown with a thick border while all other selections are shown with a dotted border, as in Figure 4-17.

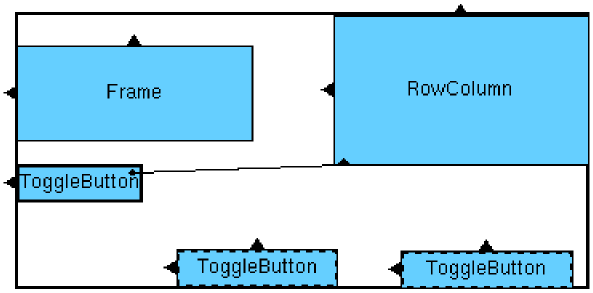


FIGURE 4-17 Primary and Secondary Selections

“Align” will align *to* the primary selection.

4. **Click with <*Shift-Button 1*> on the middle ToggleButton.**
5. **Click with <*Shift-Button 1*> on the left ToggleButton.**

   To align the tops of the ToggleButtons:

6. **Select “Align top edges” from the “Align” pull-right menu in the Layout Menu.**

   This sets the attachments shown in Figure 4-18.

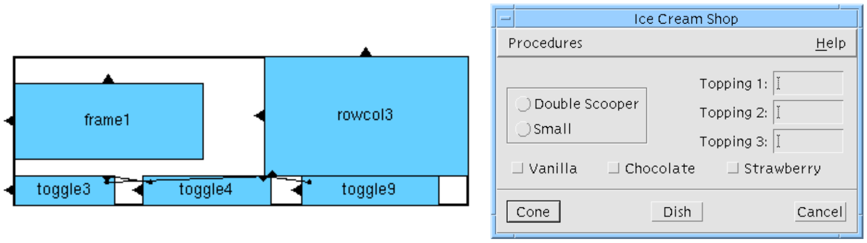


FIGURE 4-18 Layout After Aligning the Three ToggleButtons

Direction of Attachments Set by Group Alignment

When you align a group of widgets, the last widget selected is unaffected and the others are aligned to it. The order in which widgets are selected does not matter except for the last. Sun WorkShop Visual sets attachments in the order of the widget’s spatial positions. Each widget in the group except the last is attached to its neighbor and all attachments point toward the last widget selected. In your layout, the right widget is attached to the center widget, which is attached to the left widget.

Figure 4-19 illustrates this general rule. If a group of widgets is selected in the order shown in the first figure, the resulting attachments would connect them in the order shown in the second figure.

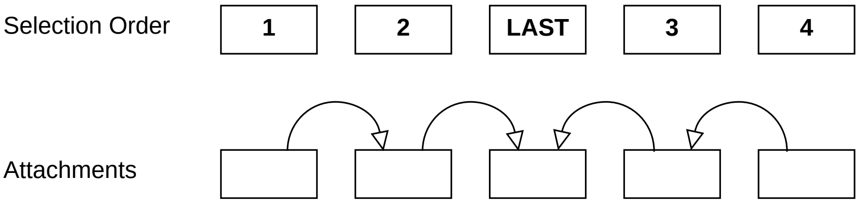


FIGURE 4-19 Direction of Attachments Set by Group Align

This rule works similarly for columns of widgets aligned along their left or right edges.

Note – As when setting other attachments, be careful to avoid circularity in aligning groups of widgets. The best method is to position the top or left-most widget where you want it and then align other widgets to it, selecting them from right to left or from bottom to top.

Distribute

The distribute functions on the “Distribute” pull-right menu in the Layout Menu let you distribute a group of widgets evenly across a given space. These functions are only enabled if you have more than two widgets selected. See “Selection: Single, Primary, and Secondary” on page 102 for details on how to select more than one widget.

The Distribute Functions

Selecting the “Distribute” item in the Layout Menu displays a pull-right menu containing two items which are also pull-right menus.

Horizontal

Selecting “Horizontal” displays a pull-right menu with the following items:

- Centres. Distributes horizontally so that the centres are equally spaced between the two farthest edges.
- Edges. Distributes horizontally so that there is an equal space between the edges within the space delimited by the two farthest edges.
- Constant. Distributes horizontally leaving a gap between widgets as specified in the “Offset” field. If the “Offset” field is empty, the default Form spacing is used. In this mode, the total space occupied by the group of widgets may change.

Note that the difference between the first two items is most visible when the widgets being distributed are very different in size.

Vertical

Selecting “Vertical” displays a pull-right menu with the following items:

- Centres. Distributes vertically so that the centres are equally spaced between the topmost and bottommost edges.
- Edges. Distributes vertically so that there is an equal space between the edges within the space delimited by the topmost and bottommost edges.
- Constant. Distributes vertically leaving a gap between widgets as specified in the “Offset” field. If the “Offset” field is empty, the default Form spacing is used. In this mode, the total space occupied by the group of widgets may change.

Note that, as with horizontal distribution, the difference between the first two items is most visible when the widgets being distributed are very different in size.

Using the Distribute Functions

You can use these functions to distribute the three ToggleButtons evenly across the bottom of the Form.

1. **Click on the right ToggleButton.**

   If widgets are still selected from the previous section, they are deselected when you click on a new widget to start a new group.

2. **Click with <*Shift-Button 1*> on the middle ToggleButton.**

3. **Click with <*Shift-Button 1*> on the left ToggleButton.**

   The three buttons are now all highlighted. The primary selection, indicated by a thick border around the widget, is the last ToggleButton selected. See “Selection: Single, Primary, and Secondary” on page 102 for more details. However, the order of selection does not matter to the “Distribute” feature.

   Distribute the buttons horizontally with equal space between their edges:

4. **Pull right from “Distribute” in the Layout Menu.**

5. **Pull right from “Horizontal” in this second menu.**

6. **Select “Edges”.**

   In this case you could equally well choose “Centres”. The result would be much the same because the selected widgets are equal in size.

   The result is shown in Figure 4-20.

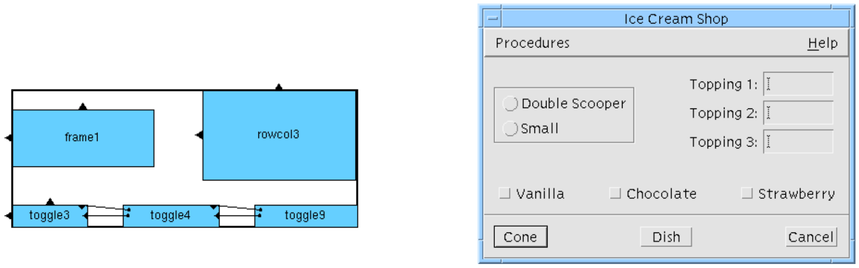


FIGURE 4-20 Distribution of Widget Edges

Direction of Attachments Set by “Distribute”

“Distribute” sets attachments in a different order from group align (“Align” in the Layout Menu). With “Distribute”, attachments are always made in spatial order from bottom to top or from right to left, as shown in the Figure 4-21. Each widget is attached to its neighbor. In “Distribute”, unlike “Align”, the order of widget selection makes no difference.

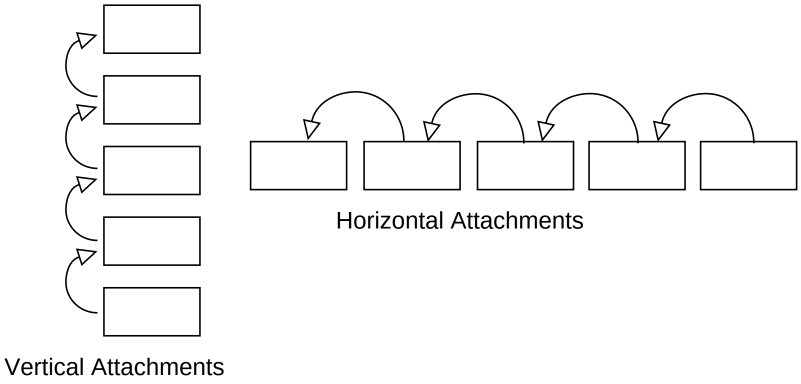


FIGURE 4-21 Direction of Attachments Set by “Distribute”

You can see in the Layout Editor screen that the arrows attaching the ToggleButtons edge to edge all point from right to left. It is easier to see the arrows if you pull down the View Menu and temporarily remove class name annotations from your screen.

Avoiding Circularity Tip

Because “Distribute” only sets attachments as described above, circular attachments can easily result unless you plan ahead. For example, you can select widgets from right to left, “Align” them and then “Distribute” them, as you have just done. However, you cannot select them from left to right, “Align” them and then use “Distribute” on the same group of widgets. Doing this results in circular attachments.

If you make all other attachments from right to left or from bottom to top, “Distribute” never results in circularity.

The attachments set by “Align” and “Distribute” may also conflict with existing attachments - see “Removing Attachments on Move” on page 108 for more details.

Proportional Spacing: the Position Mode

Position attachments let you attach an edge of a widget at a position that is a percentage of the Form’s total width or height. This capability lets the widgets in your interface take advantage of additional space when the window resizes. Positions are always measured from the top left corner of the Form. If the top or bottom edge of a widget has a position attachment of *n%*, that edge is positioned *n%* of the way down from the top of the Form. If the left or right edge of a widget has a position attachment of *n%*, that edge is positioned *n%* of the way across from the left edge of the Form.

The position is specified as a percentage value in the “Position” field. If you do not enter a value, Sun WorkShop Visual assumes a value of zero. Position attachments are shown as hollow circles on the attached edge of the widget.

First, demonstrate the current window behavior:

1. **Resize the window so that it is wider than the present size.**

   Note that the RowColumn stays at the same distance from the Frame, as shown in Figure 4-22.

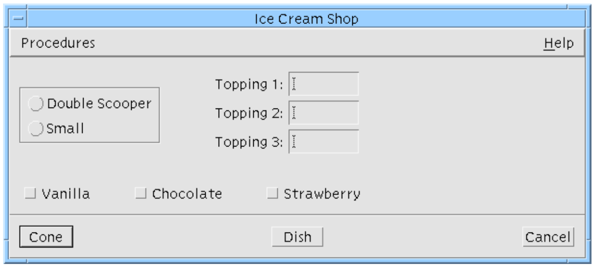


FIGURE 4-22 Behavior of RowColumn with Fixed Offset Attachment When Window is Resized

This is the behavior you expect when you set an attachment with a fixed offset. Any extra window width is just unused space. Many interfaces use this resize behavior. However, you can also use a position attachment to make the RowColumn move over to take advantage of available window space.

To do this, specify a 45% position attachment on the left edge of the RowColumn:

2. **Click on the "Position" toggle.**
3. **Double-click in the "Position" box and type: 45**
4. **Position the pointer just inside the left edge of the RowColumn so that the edge highlights and click.**

   The position attachment appears as a hollow circle on the edge of the RowColumn, replacing the existing attachment and the arrow that represented it. This type of attachment places the RowColumn's left edge 45% of the distance across the Form, regardless of the window size. To see the effects of this:

5. **Resize the window narrower, then wider.**

   The RowColumn now moves right to fill any extra space, as shown in Figure 4-23. This type of resize behavior is the main advantage of position constraints.

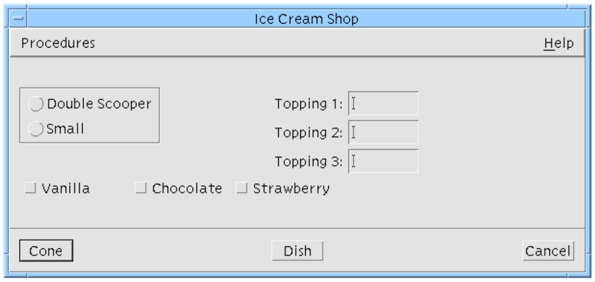


FIGURE 4-23 Behavior of RowColumn with Position Attachment When Window is Resized

The disadvantage of this type of attachment is that a position attachment is calculated only by the size of the Form and does not adjust to fit the sizes of other widgets. This is a problem if other widgets resize, as shown in Figure 4-24. When one of the labels inside the Frame becomes longer, the Frame can get closer to the RowColumn, or even overlap it.

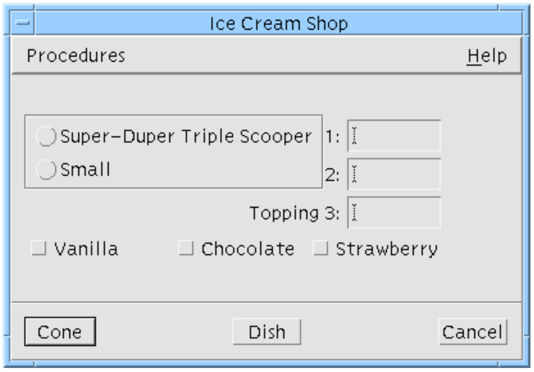


FIGURE 4-24 Behavior of RowColumn with Position Attachment When Another Widget Resizes

Compare this behavior to that shown in Figure 4-10, where the layout had a widget attachment with a 50-pixel offset. The choice of attachment type is up to you and should be based upon the types of widgets in your layout and any possible changes at run time. Various aspects of layout strategy are discussed further throughout Chapter 20, “Advanced Layout”, starting on page 567.

Self Mode

The “Self” mode is another way of setting a position attachment. Instead of typing a percentage value, you click on one edge of the widget and Sun WorkShop Visual calculates a percentage based on the widget’s present location and the present size of the Form. When you use “Self”, you do not have to specify a percentage in the “Position” field and any value that is already in the field is ignored. You must, however, first place the widget where you want it to be using “Move” or one of the other commands.

“Self” works especially well in combination with “Distribute”. You have already set up the buttons at the bottom of the Form with a fixed gap between them. By setting “Self” attachments on both sides of each button, you can preserve the evenness of spacing while letting the buttons take advantage of extra window space that may become available.

By default, “Self” attachments snap to the grid. Therefore, in order to take advantage of the precise spacing set by “Distribute”, you should disable the grid.

1. **Set the grid slider to 0.**

2. **Click on the “Self” toggle.**

   In “Self” mode, positions are calculated relative to the total size of the Form. Since you have been changing the window size, you should:

3. **Reset the Form.**

   Resetting the Form calculates its best size based on the attachments currently set on its children.

4. **Click on the right edge of the right ToggleButton.**

   The “Self” attachment appears as a filled circle on the edge of the ToggleButton.

5. **Click on the left edge of the right ToggleButton.**

6. **Click on the right edge, then the left edge, of the middle ToggleButton.**

7. **Click on the right edge, then the left edge, of the left ToggleButton.**

8. **Reset the Form.**

   “Self” attachments appear as filled circles. In Motif, however, a “Self” attachment is the same as a position attachment and therefore Sun WorkShop Visual cannot tell them apart after you reset the Form. In this event, “Self” attachments appear as hollow circles and behave exactly like Position attachments. After resetting the Form, the ToggleButtons look as shown in Figure 4-25.

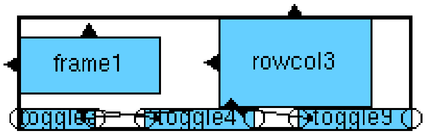


FIGURE 4-25 “Self” Attachments after Form Reset

9. **Save your design.**

   This concludes the tutorial portion of this chapter. The rest of this chapter discusses some additional layout features.

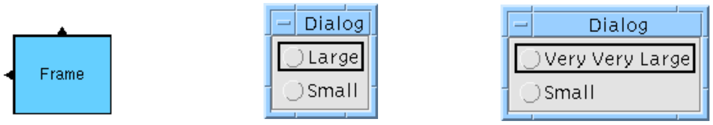
Resize Mode

"Resize" works like "Move" but sets attachments on the bottom and right side of a widget. To use "Resize":

1. **Click on the "Resize" toggle.**
2. **Using mouse button 1, drag the lower right corner of a widget to the position you want.**

"Resize" is useful with BulletinBoards and DrawingAreas if you want to fix the size of a widget. In a Form, "Resize" works by attaching the lower right corner of the widget to a specific *x,y* location relative to the upper left corner of the layout widget. When combined with attachments on the upper left corner of the widget, this fixes the widget's size. In a BulletinBoard or DrawingArea, "Resize" simply sets the width and height resources of the widget.

This option is not normally used with Form layouts, because most widgets behave better when you let them calculate their own best size. Figure 4-26 shows a typical example of how widgets can resize themselves.



"Move" attachment controls position of Frame's upper left corner

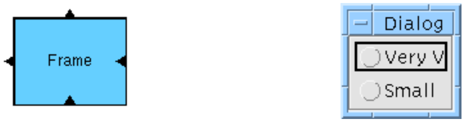
Frame with original set of labels

Frame resizes when ToggleButton labels get longer

FIGURE 4-26 Frame Widget Resize Behavior

The Frame in Figure 4-26 is constrained by a "Move" command only. If the user changes the label text for one of the ToggleButtons, the Frame is free to resize itself because its bottom and right sides are unconstrained.

If the Frame also has attachments set by "Resize", however, it cannot resize, as shown in Figure 4-27.



Resize attachment also fixes Frame's lower right corner

Frame is not allowed to resize when labels get longer

FIGURE 4-27 Effects of Using Move and Resize Attachments Together

Because the combination of "Move" and "Resize" attachments shown in Figure 4-27 fixes all four sides of the Frame, it cannot subsequently expand to accommodate a larger label. Motif handles this situation by displaying only part of the label string.

"Resize" is used mainly with the BulletinBoard and DrawingArea as these widgets do not offer position attachments or widget attachments. "Resize" offers a way to force widgets to remain at a certain size and prevents them from overlapping. The disadvantage of "Resize" is that it eliminates the positive effects of automatic resizing.

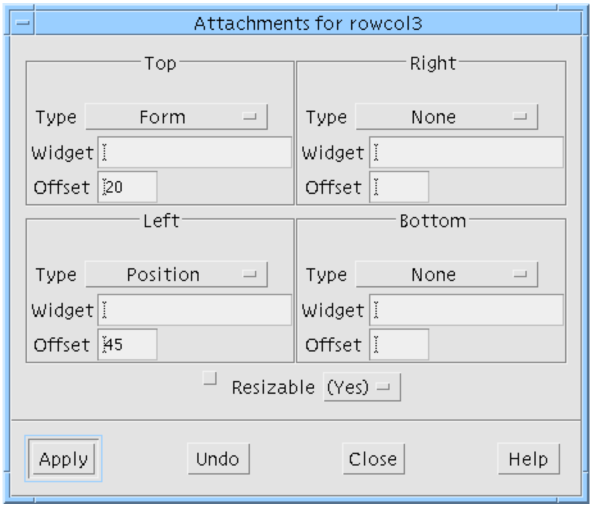
To get the best behavior with widgets that are likely to change size, use a Form and attach widgets to one another so that when one widget changes size other widgets move to accommodate it.

Using the Constraints Panel

The constraints panel, which was introduced in the previous chapter, can be used to view attachments on any child of the Form and to adjust attachments. The constraints panel is only recommended for viewing or fine-tuning attachments. Note that the constraints panel only shows attachments that originate from a widget. Use the Layout Editor for any large-scale changes.

1. **Select "Close" from the File menu on the Layout Editor screen.**
2. **Select the RowColumn in the construction area.**
3. **Pull down the Widget Menu and select "Constraints".**

This command displays the attachments set on the RowColumn, as shown in Figure 4-28.



Attachments for rowcol3

Top

Type Form

Widget

Offset 20

Right

Type None

Widget

Offset

Left

Type Position

Widget

Offset 45

Bottom

Type None

Widget

Offset

□ Resizable (Yes)

FIGURE 4-28 Constraints Panel for RowColumn

The top edge of the RowColumn has an attachment to the Form with an offset of 0. The left edge of the RowColumn has a position attachment of 45%. The right and bottom edges of the RowColumn have no attachments.

You can use the constraints panel to:

- View the attachments on any child of a Form
- View the names of widgets to which each edge of the widget is attached
- Distinguish default offsets (shown in parentheses) from explicit offsets
- Change the type of attachment - “Form”, “Position”, or “Widget” - by selecting from the “Type” option menu
- Remove an attachment by selecting “None” from the “Type” menu (only if the widget still has at least one attachment in *x* and one in *y*)
- Adjust the offset or position by typing a new value into the “Offset” field
- Specify whether the Form should resize if this child resizes. This is the “Resizeable” option menu. If “No” is selected here, the Form parent will remain the same size whatever size this child becomes.

4. **To effect any changes in the constraints panel, click on “Apply”.**

Other Layout Widgets

The Layout Editor also works on the BulletinBoard, Drawing Area and RowColumn widgets.

BulletinBoard and DrawingArea

The Layout Editor works in much the same way on the BulletinBoard and DrawingArea as on the Form. There are a few differences when you use it with other layout widgets. The following comments refer only to the BulletinBoard and DrawingArea.

As mentioned above, widget, self and position attachments are not available. The Layout Editor does not show arrows or arrowheads indicating attachments, but only the positions and sizes of widgets in the layout.

When you first display the Layout Editor, you may find that several widgets are initially laid out directly on top of one another. Use “Move” repeatedly to drag them to new positions so that they do not overlap.

The only available editing modes on the main screen are “Move” and “Resize”. You can also use “Group Align” and “Distribute” but not the “Align” mode on the main screen. “Group Align” and “Distribute” do not attach widgets to one another, but merely reposition them.

There is no danger of circular attachment as widgets cannot be attached to each other.

Internally, the Layout Editor does not set resources of the BulletinBoard and DrawingArea as it does for the Form. Instead, it determines layout by setting Core size and position resources of the child widgets. The constraints panel is not available for these layout widgets. To view the size and position resources, display the Core resource panel for the child widget.

RowColumn

The RowColumn is a manager widget and, therefore, the Layout Editor can be invoked. The only function you can use, however, is “Resize”. None of the others apply to RowColumns.

Layout Editor Restrictions

You cannot move a widget in the layout editor, or set constraints on it, if:

- it is not managed
- it is part of a definition instance and you have not been given access to it (see "Modifying and Extending an Instance" on page 273)

You can manage a widget by setting the "Managed" toggle for it on the Code Generation page of the Core resources dialog.

CHAPTER 5

Other Editors

Introduction

Sun WorkShop Visual provides special editors which simplify certain common tasks. You can use these editors to perform the following tasks:

- Setting colors
- Setting fonts for text strings
- Using bitmaps or pixmaps for labels instead of text
- Creating pixmaps
- Creating and editing compound strings
- Editing callbacks and preludes

The editors are used in this chapter to customize the tutorial interface. They are invoked from the widget resource panels and the result added to the appropriate field in the resource panel. It is also possible to display the Color Selector, Pixmap Editor and Font Selector from the "Tools" menu. The editors can be used independently of any selected widget.

Setting Colors

Because foreground and background colors are the basic elements of all widgets, these resources are located on the Core resource panel. Use the following instructions to set these colors for widgets in the hierarchy.

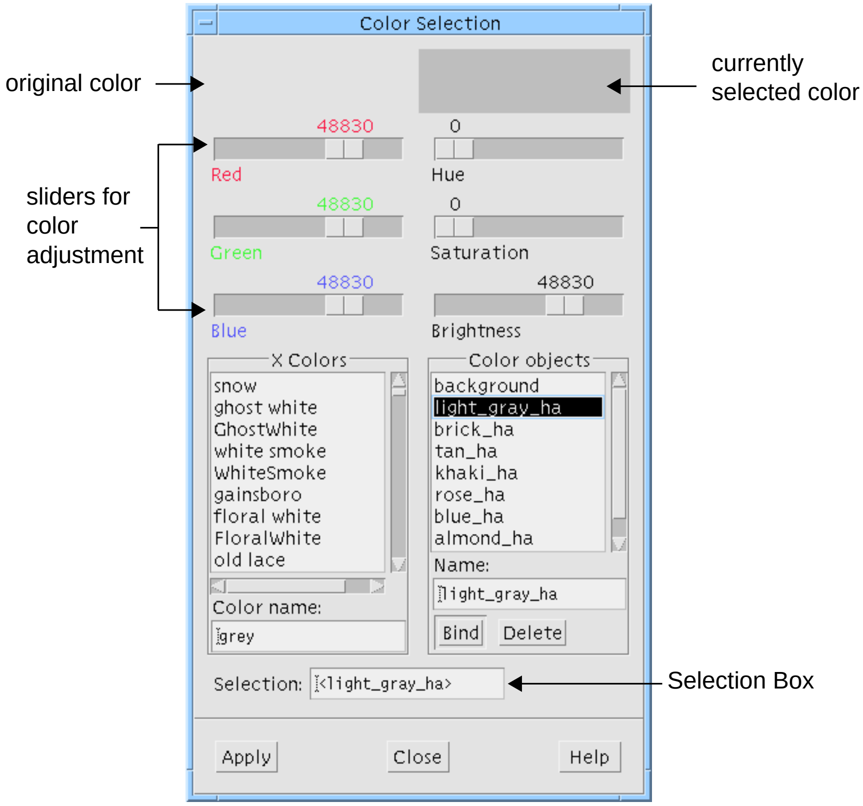
1. **Select the "Strawberry" ToggleButton icon in the hierarchy.**

2. **Click on the “core resources” button on the toolbar or pull down the Widget Menu and click on “Core resources...”.**

   In previous chapters, you entered settings for resources directly in the text boxes on the right side of the resource panels. To use the editors described in this chapter, click on the buttons on the left side of the resource panels instead.

3. **On the “Display” page, click on the “Foreground color” PushButton.**

   Sun WorkShop Visual displays the Color Selector, shown in Figure 5-1:



Red: 48830
Green: 48830
Blue: 48830
Hue: 0
Saturation: 0
Brightness: 48830
Color name: grey
Name: light_gray_ha
Selection: <light_gray_ha>

FIGURE 5-1 The Color Selector

Selecting from the X Colors List

X provides many pre-named colors. These standard colors make a good starting point for selecting colors for the interface.

1. **Scroll down through the X colors displayed in the scrolled list.**

2. **Select a color.**

   As you select colors, the currently selected one is shown on the right at the top of the dialog. On the left, the original color is displayed so that you can compare the two.

   Choose a dark reddish color such as maroon. These colors are clustered about halfway down the list. The selected color is displayed at the top of the Color Selector.

3. **Click on "Apply" in the Color Selector.**

   This applies your selection to the "Foreground color" resource on the Core resources panel. To apply it to the widget, you must:

4. **Click on "Apply" in the Core resources panel.**

   The "Strawberry" ToggleButton changes color in the dynamic display. Don‘t worry if it looks like the background color changes instead of the foreground color; this is because the widget is selected in the hierarchy and the selection is reflected in the dynamic display by inverting the foreground and background colors.

   To see the true colors:

5. **Select the Shell in the hierarchy.**

   Now you can see the true foreground and background colors of the "Strawberry" ToggleButton in the dynamic display.

Using Color Components

Selecting from the X colors list is only one of several ways to specify a color. Another method is to create the color using components. Use this technique to set the background color of the "Strawberry" ToggleButton:

1. **Select the "Strawberry" ToggleButton in the hierarchy.**

2. **On the Core resource panel, click on "Background color".**

3. **Use the sliders to change the color.**

   The sliders at the top of the Color Selector let you individually control the red, green and blue components and the hue, saturation and brightness of your color. You can use the sliders in any order. Changes are reflected immediately at the top of the Color Selector. Notice that the color name is a concatenation of values.

   Since this is a background color, a light (non-saturated) color is recommended. This provides a good contrast for the labels, which are darker.

4. **When you are satisfied with the color, apply it by repeating Step 3 and Step 4 in "Selecting from the X Colors List" on page 136.**

   Do not change the color in the Color Selector before proceeding to the next section.

Color Objects

To create a visually appealing interface, it is essential to use colors consistently. Sun WorkShop Visual assists you by providing *color objects*, which let you bind colors to names which you specify. Use this feature to apply the same background color to all the widgets in the central part of the tutorial interface:

1. **Select the "Strawberry" ToggleButton in the hierarchy.**
2. **In the Core resource panel, click on "Background color".**
3. **In the Color Selector, click in the "Name" Text box underneath the list of Color objects.**
4. **Type: `background`**
5. **Click on "Bind".**

   This binds the color to a color object named "background". The selection box shows the name *background* in angle brackets.

6. **Click on "Apply" in the Color Selector and in the Core resources panel.**

   Apply this background color to other widgets in the hierarchy by entering the color object name as the setting for the background color resource:

7. **Select the "Vanilla" ToggleButton in the hierarchy.**
8. **In the Core resource panel, double-click in the "Background color" text field.**
9. **Type: `<background>`**

   Note that you must use angle brackets. The angle brackets distinguish an object name from a string value. For example, a color object named *<red>* is distinct from the color *red*.

10. **Click on "Apply".**

    The color in the "Vanilla" ToggleButton changes. Now apply this background color to the "Chocolate" ToggleButton and any other widgets you want to share the same background color.

Rebinding Color Objects

When you use a color object, you can easily change the color on all widgets which reference that color object. This makes experimenting with colors easy.

1. **In the Color Selector, click on "background" in the "Color objects" list.**

The background color is displayed at the top right of the Color Selector and “background” is displayed in the text field underneath the list of objects.

2. **Using the sliders at the top of the Color Selector, change the color.**

3. **Click on “Bind”.**

   All the resources which refer to the color object change to the new color and the change is reflected immediately in the dynamic display.

   Experiment with creating new colors, binding them to color objects and assigning these color objects to some color resources. You can also bind colors from the X colors list to color objects. By repeating these steps, you can build your own palette of colors. Remember that it is better to name a color object for the function it serves, such as “background”, than for the color it represents, since the color can change.

   Color objects are saved with the design file. This means you can use the same names for color objects in different design files, even though the colors might be different. For example, the color object, *background*, might be yellow in one design and light blue in another. Within the same design, however, an object name such as “background” always refers to the same color.

   Color objects can be shared between files by making them “global”. This is controlled by setting the appropriate toggle when you generate code. See “Global Object Functions” on page 219 for more information.

Setting Fonts

The Font Selector lets you select font styles and sizes for the text which appears in your widgets. Your system determines which font styles are available; you can’t create new fonts the way you can create new colors.

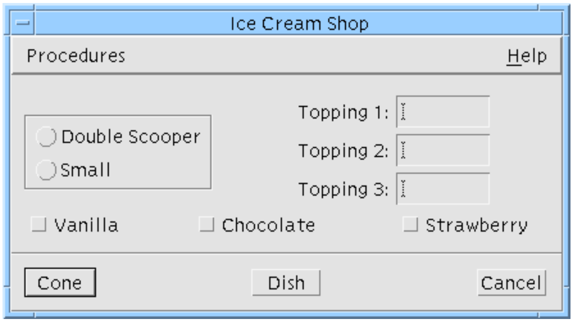
So far, you have used the default font for all text in the tutorial interface. In this section you will use the Font Selector to:

- Select a font
- Set a font on a single widget
- Bind a font object to a particular font
- Apply a font object to multiple widgets

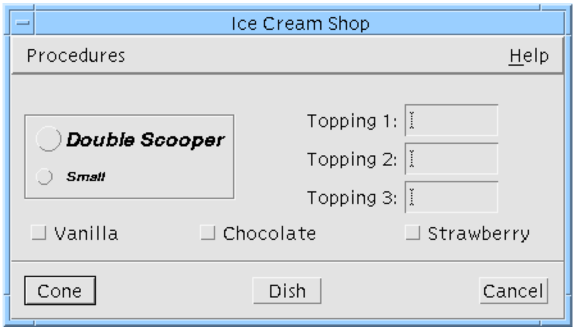
This section discusses the use of a single font in a text string. Complex font objects which let you use multiple fonts in a single label string are discussed in “Compound Strings” on page 163.

Selecting a Font

To add more visual interest to the tutorial interface, you are going to change some of the fonts from their default to an oblique font, as illustrated in Figure 5-2.



Default font



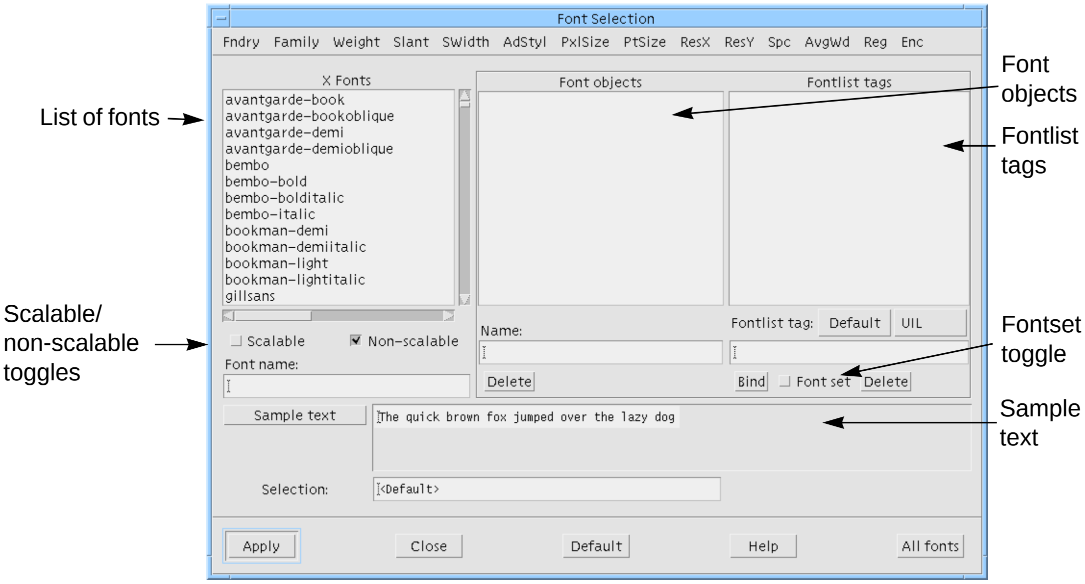
Oblique font

FIGURE 5-2 Toggle Buttons Before and After Setting Font

First, bring up the Font Selector:

1. **Click twice on the "Double Scooper" radio button to bring up the resource panel.**
2. **On the "Display" page of the resource panel, click on "Font".**

Sun WorkShop Visual displays the Font Selector, shown in Figure 5-3.


Font Selection
Fndry Family Weight Slant SWidth AdStyl PxlSize PtSize ResX ResY Spc AvgWd Reg Enc
X Fonts
Font objects
Fontlist tags
Name:
Fontlist tag: Default UIL
Font name:
Delete
Bind
Font set
Delete
Sample text
Selection:
Apply Close Default Help All fonts
List of fonts
Font objects
Fontlist tags
Scalable/ non-scalable toggles
Fontset toggle
Sample text

avantgarde-book
avantgarde-bookoblique
avantgarde-demi
avantgarde-demioblique
bembo
bembo-bold
bembo-bolditalic
bembo-italic
bookman-demi
bookman-demiitalic
bookman-light
bookman-lightitalic
gillsans

□ Scalable ☑ Non-scalable

□ Font set

Sample text: The quick brown fox jumped over the lazy dog

Selection: <Default>

FIGURE 5-3 The Font Selector

Regions of the Font Selector Panel

The Font Selector lists all the fonts available on your system. Because different machines may have different fonts installed, your list may look different from the figure.

Since there can be hundreds of fonts in the list, the menu bar lets you filter the list according to different criteria such as the font family, weight and point size.

The toggles below the font list let you select scalable fonts, non-scalable fonts, or both.

When you select a font from the list, the name appears in the “Font name” and “Selection” fields.

The Font Selector also lists font objects and their associated fontlist tags. Simple font objects can be used as aliases for font names. “Compound Strings” on page 163 discusses the use of more complex font objects.

Filtering the Font List

You are going to change the font of the “Double Scooper” label to 14-point bold oblique Helvetica. (If this font is not installed on your machine, select an alternative.) Because there are so many fonts in the list, it helps to filter the list. The “Font name” and “Selection” fields reflect your choices as you make them.

1. **Pull down the Family Menu from the menu bar and select “Helvetica”.**

   This eliminates all fonts from the list which are not Helvetica.

2. **Pull down the Weight Menu and select “bold”.**

3. **Pull down the Slant Menu and select “o” for “oblique”.**

4. **Pull down the PtSize (point size) Menu and select “140”.**

   Note that, because point sizes are specified in tenths of a point, this selects a 14-point font.

   These steps reduce the font list to about four entries. A typical font name entry is shown in Figure 5-4. Although many fields are required to specify a font uniquely, most of them are only interesting to advanced typographers. The fields you are most likely to use are family, weight, slant and point size. You may also need to specify the display resolution such as 75 dpi (dots per inch) or 100 dpi. These fields are identified in Figure 5-4.

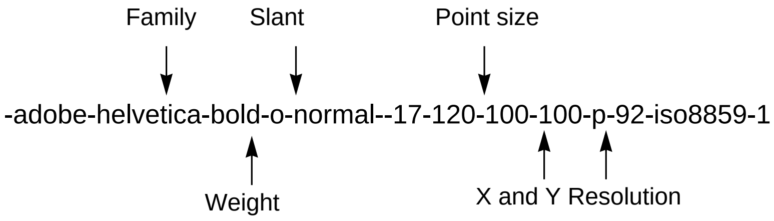

Family
Slant
Point size
-adobe-helvetica-bold-o-normal--17-120-100-100-p-92-iso8859-1
Weight
X and Y Resolution

FIGURE 5-4 A Typical Font Name

The example in Figure 5-4 specifies a display resolution of 100 dpi, both horizontal and vertical, which is appropriate for most workstation displays.

5. **Pull down the ResX Menu and select a resolution of 100 or 75 dpi.**

   Select the resolution appropriate to your display. If in doubt, use 100 dpi.

Applying the Font

The selections you have made are sufficient to specify a font for the interface. Now you can apply it to the “Double Scooper” radio button.

1. **Click on “Apply” in the Font Selector.**

   This applies your selection to the “Font” resource in the resource panel.

2. **Click on “Apply” in the resource panel.**

   The “Double Scooper” radio button is now labeled in a large, bold, oblique font.

3. **Select the “Small” radio button.**

4. **Pull down the PtSize (point size) Menu and select “100”.**

   You can press the “Sample text” button to re-display the example text whenever the filtering has been changed. Sun WorkShop Visual doesn’t do this automatically as some fonts can take a very long time to load.

5. **Click on “Apply” in the Font Selector and in the resource panel.**

   The “Small” radio button is now labeled in a bold, oblique font, smaller than the “Double Scooper” label. Before proceeding, reset the Font Selector panel so that it shows all the fonts:

6. **Click on the “All fonts” button at the bottom of the Font Selector.**

   This resets all elements of the font filter to “*” and so all available fonts are displayed again.

Scalable Fonts

The font you used on the radio buttons is a non-scalable font. This means that it is only available in certain fixed point sizes. X also supports scalable fonts, which can be any size you like. Try selecting some scalable fonts:

1. **Clear the “Non-scalable” toggle in the Font Selector.**

   This eliminates all non-scalable fonts from the list and so the list is now empty.

2. **Set the “Scalable” toggle.**

   This adds the scalable fonts to the list. You can specify a size for any of these fonts in two ways: by adjusting either the pixel size or the point size. The pixel size is the first numeric field in the font descriptor and the point size is the second numeric field. Both of these fields are initially set to zero for all scalable fonts, indicating a default size.

3. **Click on one of the font names in the list.**

   Your selected font appears in the "Font name", "Selection" and "Sample text" fields.

4. **Click in the "Font name" field (not the "Selection" field).**

5. **Edit the second numeric field (point size) to 240 (24 points).**

6. **Press** *<Return>* **or click on the "Sample text" button to display a sample of this text size.**

   When using scalable fonts, specify a non-zero value for either the pixel size or the point size, but not both. X adjusts the remaining zero value to fit the explicitly specified value. If you specify non-zero values for both fields, however, Sun WorkShop Visual displays an error message if they do not match.

   Typical point size values, specified in tenths of a point, are larger than typical pixel size values. A point size of 240 roughly corresponds to a pixel size between 30 and 35. The exact proportion depends on the resolution of your screen and the specific font.

   X has two ways of scaling fonts: outline scaling and bit scaling. If the sample text is very jagged, the font is bit-scaled. To list only outline-scaled fonts, pull down the Fndry Menu and select "bitstream".

   After experimenting, set the Font Selector to see the non-scalable fonts again:

7. **Click on the "Non-scalable" toggle.**

Simple Font Objects

For the two radio buttons, you set the fonts individually. If you use this method to set the same font for multiple widgets, any later changes must also be made for each widget individually. Also, the code generated by Sun WorkShop Visual for the application makes a separate call to your system to load the same font for each label, which is inefficient.

Therefore, if multiple widgets use the same font, you can simplify both the code and maintenance of it by creating a simple font object. A *font object* is an alias for a list of fonts. A *simple font object* is an alias for a one-element list.

1. **Use the pulldown menus to select the 12-point bold oblique helvetica font.**

   If more than one font appears in the list of X fonts, highlight one.

2. **In the "Name" field underneath the list of Font objects, type**: `option_labels`

   Remember that it is better to name a font object for the function it serves rather than for the size or style it represents, since these specifications can change.

3. **Click on “Bind”.**

   This creates a font object called “option_labels”. It only has one font in its list: the 12-point helvetica font. This has an associated fontlist tag “<Default>”. The use of fontlist tags is discussed in “Compound Strings” on page 163.

   Notice that the “Selection” field automatically updates to show the “<option_labels>” name. The angle brackets (<>) indicate that it is a font object rather than a font name. This font is applied to the resource panel when you click on “Apply”. Do not do this yet as we are going to apply this font object to all the ToggleButtons and Labels at once.

4. **Select all three ToggleButtons: “Vanilla”, “Chocolate” and “Strawberry” and the three Labels: “Topping1”, “Topping2” and “Topping3”.**

   Do this by either dragging a rectangle around the widgets or by selecting each widget while holding down the Shift key.

5. **Click on “Apply” in the Font Selector.**

   This applies the font object to the ToggleButton resource panel.

6. **Click on “Apply” in the resource panel.**

   This applies the font object to all the selected widgets.

   The interface now looks as shown in Figure 5-5.

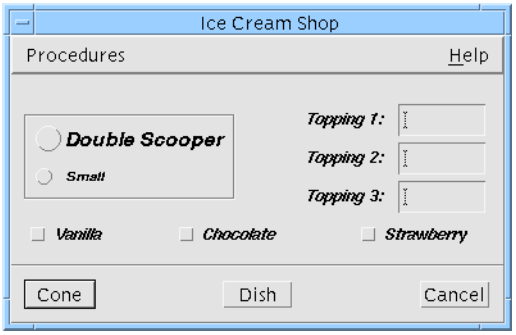


FIGURE 5-5 Interface with Fonts Applied

Changing the Font Object

The text on the ToggleButtons and Labels is now shown in the font to which the font object is bound. To change this font style, just bind the font object to another font.

1. **Pull down the Slant Menu and select “r” for “regular”.**

2. **Pull down the Family Menu and select “Times”.**

   The font list now displays the 12-point bold Times fonts. If the list is empty or Times is not availab, select a different font family.

3. **Click on “Bind”.**

4. **Click on “Apply” in the Font Selector.**

5. **Click on “Apply” in the resource panel.**

   The font object changes to correspond to the Times font and all the Labels and ToggleButtons change immediately in the dynamic display.

   Font objects are saved with the design file. This means you can use the same names for font objects in different design files, even though the font might be different. For example, the *option_labels* font object might be Helvetica in one design and Times in another.

   Font objects can be shared between files by making them “global”. This is controlled by setting the appropriate toggle when you generate code. See “Global Object Functions” on page 219 for more information.

Selecting Pixmaps

You can use pixmaps instead of text strings on labels and buttons. Sun WorkShop Visual provides two editors for creating and applying pixmaps. First you will use the Pixmap Selector to learn the basics of applying pixmaps to widgets. Then you will use the Pixmap Editor to create some custom pixmaps.

Sun WorkShop Visual lets you use bitmaps created using the standard X bitmap editor. It also lets you use pixmaps in the public domain *Xpm* format and provides an editor for you to build pixmaps in this format. The *Xpm* library is included with the Sun WorkShop Visual release.

Bitmaps and Pixmaps

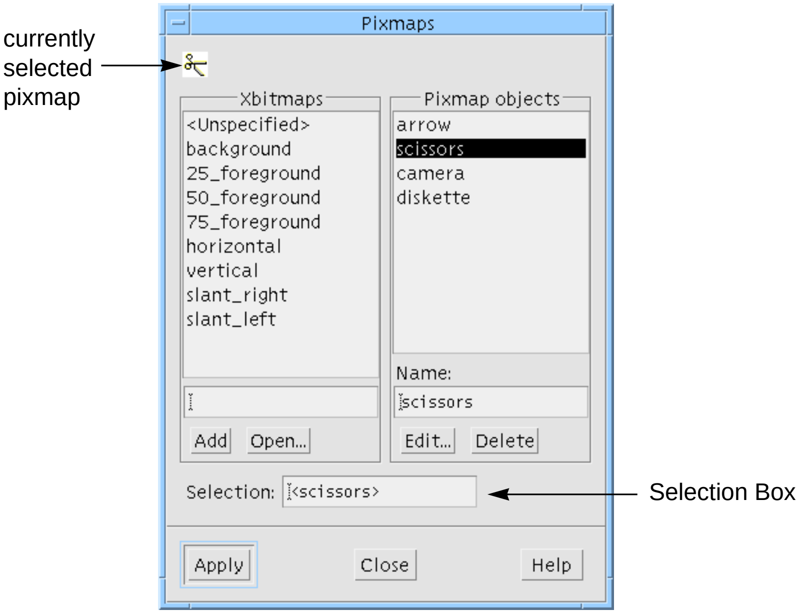
Note that you can use pixmaps created with any other utility provided they are in *Xpm* format. The difference between a bitmap and pixmap is that bitmaps are monochrome and pixmaps are color images. They are also different formats - bitmaps are *Xbm* format and pixmaps are *Xpm* format. You can still edit X bitmaps using the Sun WorkShop Visual Pixmap Editor. When you do this, Sun WorkShop Visual converts the bitmap to a two-color pixmap and writes it out in *Xpm* format. You can keep it with only two colors or add more colors to it.

Selecting a Bitmap

As a first step, replace the label of a ToggleButton with one of the X bitmaps.

1. **Click twice on the “Cone” PushButton in the hierarchy.**
2. **On the “Display” page of the resource panel, click on “Pixmap”.**

   Sun WorkShop Visual displays the Pixmap Selector. This is shown with example entries in Figure 5-6:



<Unspecified>
background
25_foreground
50_foreground
75_foreground
horizontal
vertical
slant_right
slant_left

arrow
scissors
camera
diskette

Name: scissors

Selection: <scissors>

FIGURE 5-6 The Pixmap Selector with Example Entries

3. **Select any bitmap from the list of X bitmaps.**

   The selected bitmap is displayed at the top of the Pixmap Selector.

4. **Click on “Apply” in the Pixmap Selector.**

   This applies your selection to the “Pixmap” resource in the resource panel.

5. **Click on “Apply” in the resource panel.**

   Now the ToggleButton has both a text label and a pixmap label, although only the text label appears in the dynamic display. To display the pixmap label, you must change the resource that controls which type of label is displayed.

6. **On the “Settings” page of the resource panel, change the “Type” setting to “Pixmap”.**

7. **Click on “Apply” in the resource panel.**

   The ToggleButton now displays the pixmap instead of the text label. Since the pixmap does not convey any useful information in this case, change the “Type” resource back to “String”.

   If you have additional X bitmap files on your system, you can also use these. Clicking on “Open...” displays a file selector which lets you locate bitmap files and add them to the list of X bitmaps.

   You can also enter names of bitmaps in the text box under the list of X bitmaps, then click on “Add” to add the name to the list. If the bitmap doesn’t exist yet, you can still add its name to the list and apply it to resources. Later, in development or at run time, you can supply the bitmap.

Selecting a Pixmap

The list on the right of the Pixmap Selector is the list of currently defined pixmap objects. These are pixmaps which have been bound as an object. The names in the list are the names of the objects. You can name a new pixmap object and then create a pixmap or read in an existing pixmap or bitmap. To make a new object, type its name in the text box below the list. To identify a pixmap with that object, press the “Edit” button. “Editing Pixmaps” on page 148 describes how to read in or create a pixmap.

Editing Pixmaps

Sun WorkShop Visual provides an editor for creating pixmaps. In addition to allowing you to design your own pixmaps, the Pixmap Editor also lets you:

- Bind pixmaps to pixmap objects
- Drag pixmaps and pixmap filenames into the Pixmap Editor design area
- Load pixmaps files in Xpm format
- Load X bitmap files and convert them to pixmaps for editing
- Write pixmaps to files in Xpm format

You can choose whether to write the pixmap in Xpm version 1 or 3. You should use version 3 as this is the most recent. Version 1 is provided for compatibility with older pixmap-handling applications.

All pixmaps used by widgets in Sun WorkShop Visual must be bound to pixmap objects. First, you will create a pixmap for the “Cone” PushButton, as shown in Figure 5-7:

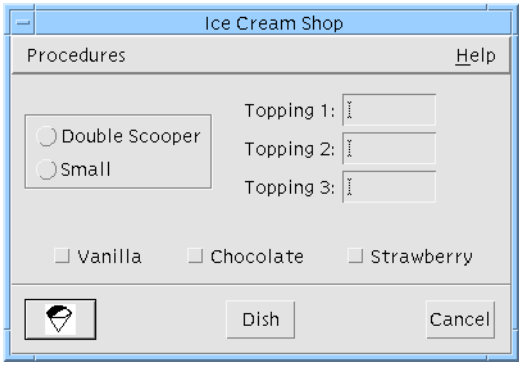
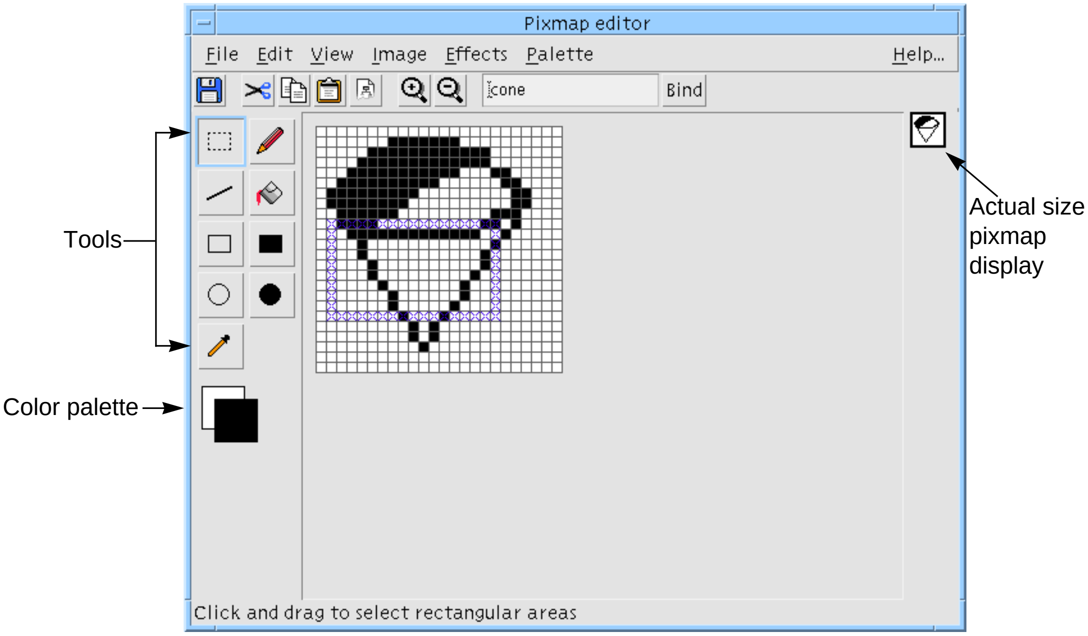


FIGURE 5-7 The Tutorial Interface with Pixmap Button

If you are continuing from “Selecting Pixmaps” on page 146, you already have the Pixmap Selector displayed and so you can skip the next two steps.

1. **Double click on the “Cone” PushButton in the hierarchy.**
2. **On the “Display” page of the resource panel, click on “Pixmap”.**
3. **In the text box in the “Pixmap objects” portion of the Pixmap Selector, type:** `cone`
4. **Press the “Edit...” button.**

Sun WorkShop Visual displays the Pixmap Editor, shown in Figure 5-8:



Click and drag to select rectangular areas

FIGURE 5-8 The Pixmap Editor

The Pixmap Editor has a menubar at the top of the window, a toolbar beneath the menubar, a tools palette on the left, the current foreground and background colors beneath the tools palette, an image drawing area in the centre and an image shown actual size on the right.

Pixmap Editor Toolbar

The toolbar, shown in Figure 5-9 contains buttons for the following operations:

- Open. Open a pixmap file, exactly as in “Open file” on page 151.
- Cut. Cut the selected area of the pixmap. See “Cut” on page 152.
- Copy. Copy the selected area of the pixmap. See “Copy” on page 153.
- Paste. Paste the contents of the clipboard into the pixmap. See “Paste” on page 153.
- Clear. Clear the selected area. See “Clear” on page 153.
- Zoom in / Zoom out. Zoom in magnifies the whole image and zoom out makes the image appear smaller. These are view options and do not affect the pixmap.

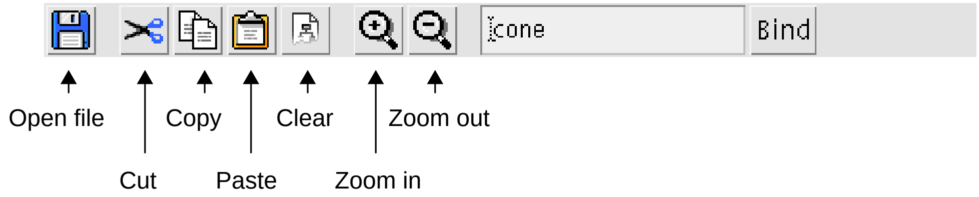


FIGURE 5-9 Pixmap Editor Toolbar

Pixmap Editor File Menu

The File menu contains four options: “New”, “Open File”, “Save XPM File” and “Close”.

New

New creates a new, blank editing area removing any image that was there before. Sun WorkShop Visual warns you if you have not bound the image since making any changes. If you wish to bind the image, select “Cancel” and then bind it, as described in “Pixmap Objects” on page 162. If you do not wish to bind the image, press “OK” and continue.

Open file

If you have X pixmap files on your system, you can also use these in your interface. Selecting “Open file...” in the File Menu displays a file selector which lets you load a pixmap file into the Pixmap Editor. Sun WorkShop Visual reads pixmaps in XPM3 format. It also reads X bitmap files and converts the bitmaps to pixmaps for editing in the Pixmap Editor.

Save XPM file

Selecting “Save XPM file...” lets you write the pixmaps you create to files. Sun WorkShop Visual writes XPM1 and XPM3 format. You should normally save pixmaps in XPM3 format. XPM1 is provided for compatibility with older versions of third-party pixmap-handling utilities.

Close

This closes the editor window. If you have made changes since last binding, Sun WorkShop Visual will issue a warning. See “Pixmap Objects” on page 162 for details on binding images to objects.

Saving Your Work

Every time you bind or write to an XPM file, you effectively save the current state of the pixmap you are creating. It’s a good idea to do this frequently as you work.

Pixmap Editor Edit Menu

The Edit menu has seven items: “Undo”, “Cut”, “Copy”, “Paste”, “Clear”, “Crop” and “Select All”.

Most of these options operate on the selected portion of the pixmap. To select, use the arrow in the selection tool on the palette, as shown in Figure 5-10.

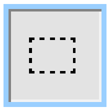


FIGURE 5-10 Selection Tool

1. **Click on the selection tool.**
2. **Click in the drawing area and drag.**

The selected portion of the pixmap includes pixels in the rectangular border marked with crosses and all the pixels within the border. Note that when you select an area of the image, the size and location of the selected area is displayed in the status line at the bottom left of the window. The “Select all” option in the Edit menu selects the entire image.

Undo

This undoes the last action in the editing area.

Cut

Cut removes the selected portion of the image to the clipboard.

Copy

Copy copies the selected portion of the image to the clipboard.

Paste

Pastes the contents of the clipboard into the selected area. If the selected area is smaller than the area on the clipboard, Sun WorkShop Visual places the top left of the clipboard image at the top left of the selected area and draws as much of the area from the clipboard as there is room to do.

Clear

Clear removes the selected portion of the image without copying it to the clipboard.

Crop

Crop reduces the image to the selected area, thereby resizing it.

Select All

This selects the whole image.

Pixmap Editor View Menu

The view menu contains one pullright menu item, labelled “Drag color”. This refers to the color of the selection rectangle as you drag it. When you pullright from this item, two radio buttons appear labelled “Invert” and “Xor”. These describe the method of showing the selection rectangle.

Pixmap Editor Image Menu

The Image menu contains one item, “Resize”. Selecting this item displays the dialog in Figure 5-11.

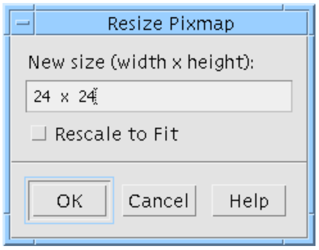


FIGURE 5-11 Resize Dialog

Type the new size in the format “width x height”. Setting the “Rescale to Fit” toggle causes the Pixmap Editor to rescale the image to suit the new dimensions specified.

Pixmap Editor Effects Menu

The Effects menu has four items: “Reflect Horizontally”, “Reflect Vertically”, “Rotate 180” and “Gray Out”.

Reflect

“Reflect Horizontally” and “Reflect Vertically” reflect the selected area across a horizontal or vertical axis respectively.

Rotate

“Rotate 180” rotates the selected area through 180 degrees - effectively reflecting the selected area across a diagonal axis.

Gray Out

This item creates a grayed out version of the selected area by ‘xor’ing each pixel. This is useful for creating insensitive pixmaps.

Pixmap Editor Palette Menu

The Palette menu allows you to edit the color palette or read in a new palette. There are two items: “Edit Palette” and “Read Palette”.

Edit Palette

This produces the Palette Editor which is described in "Color Palette" on page 157.

Read Palette

This lets you read in the palette of a saved pixmap file. When you select this item, a file selection dialog appears prompting you for the name of an *Xpm* format file. When the file is read in, the palette of colors from the saved file replaces the existing palette. Sun WorkShop Visual then changes the colors of the current image, if you have one, to use the colors of the new palette. Sun WorkShop Visual will use the closest color match it can find.

Supplied with Sun WorkShop Visual are some palettes which are color cubes. Also supplied is the palette used by Sun WorkShop Visual for its icons. These files can be found in: `$VISUROOT/lib/palettes`, where VISUROOT is the install directory of your Sun WorkShop Visual.

Tools Palette

To help you draw and color your image there is a palette of tools on the left of the Pixmap Editor window, as shown in Figure 5-12.

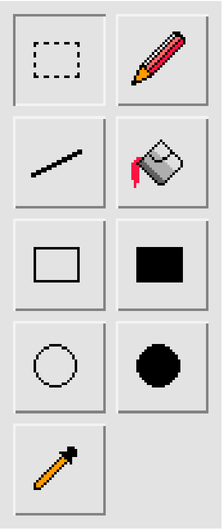
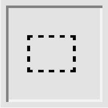


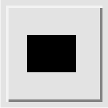
FIGURE 5-12 Tools Palette

Note that all of the drawing tools use the current foreground color. You can, however, use the current background color instead. See “Swapping Background and Foreground Colors” on page 158. You may also change all instances of a particular color in your image. See “Changing a Color in the Image” on page 158 for details on how to do this.

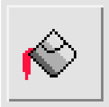
You must select one of these drawing tools to perform an operation on the window. There is always one and only one tool selected at any time.



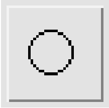
This is the selection tool. You can select rectangular areas of the image. Many of the menu functions operate on the selected area.



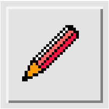
This is the filled rectangle tool. With this selected you can draw rectangles filled with the current foreground color.



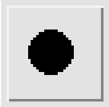
This is the fill tool. Click the mouse button over an area of the pixmap to fill that area with the current foreground color.



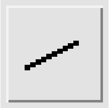
This allows you to draw the outline of a circle using the foreground color. The circle is drawn from the centre.



This enables you to draw individual pixels using the foreground color.



With this selected, you can draw a circle from the centre filled with the foreground color.



Select this to draw straight lines in the foreground color.

This is the dropper tool. Use this to *pick up* colors from the image. Clicking over a color in the image while this is the selected tool sets that color as the foreground color. See “Dropper Tool Shortcut” on page 157 for a quick way of doing this.

This allows you to draw the outline of a rectangle using the foreground color.

Dropper Tool Shortcut

With any tool selected, clicking over a color with the Control key held down sets the foreground color in exactly the same way as the dropper tool. Clicking with Mouse Button 2 while the Control key is held down sets the background color.

Dragging into the Editing Area

You can drag pixmaps and pixmap filenames into the editing area using the Motif “drag and drop” mechanism. Usually this means pressing Mouse Button 2 over the source and dragging it into the Pixmap Editor editing area before releasing the mouse button. If this does not work, check with your system administrator for your system’s configuration.

A dragged pixmap appears in the editing area exactly as if you had created it yourself, complete with a color palette containing all the colors used in the pixmap.

Color Palette

Color palettes are important in the Pixmap Editor. In order to create color pixmaps you need to specify which colors you wish to use. Any colors defined in the color palette for a pixmap will be stored with the pixmap. While you are creating your image you need to be aware which colors from the palette are the current background and foreground colors.

Background and Foreground Colors

The two squares of color below the tools palette show the current foreground and background colors. The foreground color is the color used for any drawing. The background color is used to fill blank space left by the editor. “Cut” and “Clear” both leave blank space which is filled with the background color. See “Cut” on page 152 and “Clear” on page 153 for details on these operations. “Gray out” uses the background color to produce the grayed out effect. See “Gray Out” on page 154. Resizing to a larger size fills the extra space with the background color.

Swapping Background and Foreground Colors

When you draw anything in the Pixmap Editor, the current foreground color is used. If, however, you draw with Mouse Button 2 pressed down, the current background color is used.

Seeing the Color Palette

Pressing the mouse button when the pointer is over either the background or foreground square displays the color palette.

You can edit the color palette so that it contains the colors you wish to use in your image. This is described in “Editing the Color Palette” on page 159.

Changing the Background and Foreground Colors

You can change the current background and foreground colors by displaying the color palette and releasing the mouse button over the new color.

Changing a Color in the Image

If you have a color image and you wish to change all instances of one particular color to another, do the following:

1. **Make sure the color you wish to change is the current foreground color.**
2. **Press the Control key and select another foreground color.**

Wherever the previous color was displayed in your image, the new color is now displayed.

Editing the Color Palette

Selecting “Edit palette” from the Palette menu produces the Palette Editor, as shown in Figure 5-13.

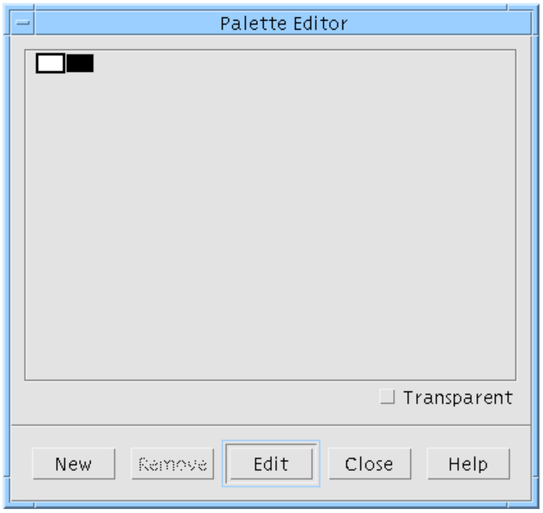


FIGURE 5-13 Palette Editor

Each color square is a button. You can select a color square by pressing the mouse button over it. The name of the selected color is displayed at the bottom of the Palette Editor window.

Removing Colors

The “Remove” button removes the currently selected color from the palette. If that color was being used in the pixmap, it is replaced by the background color.

Adding Colors

Pressing the “New” button adds a color to the end of the palette. The new color is the same as the last color on the palette.

Editing Colors

To change a color, double click over it or select it and press “Edit”. This invokes the Color Editor. See “Setting Colors” on page 135 for more details on the Color Editor.

Transparent Colors

If you set the toggle labelled “Transparent” for a selected color, the color will be “transparent” in the pixmap. This means that when the pixmap is displayed in the final application those areas of the pixmap which are transparent show the color that is beneath them. You may wish to use this so that the background of a button, for example, shows through parts of the pixmap.

Note, however, that it is XPM that supports transparent colors. The Motif PushButton and Label widgets provide no support for them. You could use the XPM library to translate the “none” color to the background color of the widget displaying the pixmap. The following file provides more detailed information on using the XPM library:

```
$VISUROOT/contrib/xpm/doc/xpm.ps
```

where VISUROOT is the install directory of your Sun WorkShop Visual.

If you wish to use transparent colors, some third-party widgets may provide more support.

The name “none” is displayed for the transparent color in the color palette editor.

Saving a Color Palette

When a pixmap is saved, the color palette is also saved. This means that you can create a pixmap specifically for its color palette and then load in the palette at another time (this is discussed below). Sun WorkShop Visual compresses the number of colors saved to those used in the picture. If you are creating a pixmap in order to save the color palette, simply set one pixel to each of the colors in your palette.

Reading a Color Palette

See “Read Palette” on page 155 for details on reading in a color palette from a saved pixmap.

Note – If you plan to use a lot of colors you should run Sun WorkShop Visual with the “-L” command line switch, as described in “Command Line Switches for Interactive Use” on page 685, so that Sun WorkShop Visual uses its own private colormap. Otherwise, you may not be able to use many colors.

Using the Pixmap Editor

First, set the size of the pixmap you’ll be creating.

1. **Select “Resize” from the Image menu.**
2. **Type: `40 x 40` into the text box in the Resize dialog.**
3. **Press “OK”.**

   The actual-size pixmap display and the grid in the drawing area should resize. To draw:

4. **Select black as the foreground color.**

   See “Changing the Background and Foreground Colors” on page 158 for details on changing the foreground color.

5. **Click on the filled rectangle tool.**

   See “Tools Palette” on page 155 if you are not sure which is the filled rectangle tool.

6. **Click in the drawing area and drag to create a black filled rectangle.**

   The actual-size pixmap display updates to show what you’ve just drawn.

7. **Experiment with the drawing tools until you feel comfortable using them.**

   It is helpful to learn the options on the Edit Menu and add some colors to the color palette before you undertake a drawing task.

8. **Add some colors to your color palette.**

   Now that you have more colors, experiment more with the drawing tools. When you are ready, use the tools and colors to create a pixmap showing an ice cream cone.

9. **Draw an ice cream cone.**

Pixmap Objects

To display your finished pixmap, you must bind it to a *pixmap object* first.

1. **Type "cone" in the "Bind" field in the Pixmap Editor.**
2. **Click on "Bind" or press <Return> in the "Bind" field.**

   In the Pixmap Selector, the pixmap object name appears in the list of pixmap objects and in the selection box.

3. **Click on "Apply" in the Pixmap Selector.**
4. **Click on "Apply" in the resource panel.**
5. **On the "Settings" page, change the "Type" setting to "Pixmap".**
6. **Click on "Apply" in the resource panel.**

   Pixmap objects work very much like color objects. You can use the same pixmap in more than one place. Changes you make to the pixmap are reflected in the dynamic display as soon as you bind again.

   Pixmap objects are saved with the design file. This means you can use the same names for pixmap objects in different design files, even though the pixmap might be different. For example, the pixmap object, *cancel*, might be a cancel stamp in one design and the international "No" symbol in another.

   If you want, try creating and adding pixmap objects for the "Dish" and "Cancel" buttons. Hint: the lettering for the "Cancel" stamp shown in Figure 5-14 was done with the line tool, not the pencil tool.

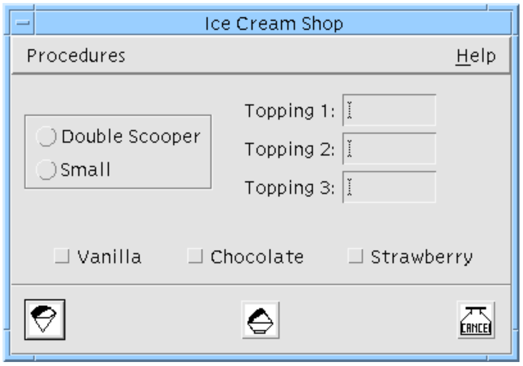


FIGURE 5-14 The Tutorial Interface with Three Pixmap Buttons

7. **Save your design.**

Pixmap objects can be shared between files by making them "global". This is controlled by setting the appropriate toggle when you generate code. See "Global Object Functions" on page 219 for more information.

Compound Strings

The labels in the tutorial example all use simple text strings. This section describes the Sun WorkShop Visual compound string editor, which uses an internal structure to let you create more complex strings. In conjunction with complex font objects, these strings let you display labels that use more than one font, or labels that are written entirely or partly from right to left.

To most people a string is just an ordered list of ASCII characters - a character string. Most Motif resources that have string values use a different string representation: the *compound string*.

Motif compound strings are used for all string resource values except for the strings in Text and TextField, which are ordinary character strings. (It is important not to confuse the Motif compound string with the compound text format of X. The Motif compound string is often called an *XmString* because this is the naming convention for the Motif toolkit functions used to manipulate it.)

A compound string is a way of encoding text so that it can be displayed in multiple languages and fonts without changing the code. In this section, you will learn how to create a string to be displayed in multiple fonts. For information about using multiple languages, see Chapter 22, "Internationalization", starting on page 615 and your Motif documentation. Appendix E, "Further Reading" is a list of recommended books on Motif.

Conceptually, a compound string consists of four types of component:

- A text string (a string of bytes)
- A fontlist tag. Fontlist tags were previously called "charsets" and this term is still used in many Motif documents
- A direction indicator: right-to-left or left-to-right
- A newline separator

Although these types of component can be in any order, it is common for each text string component to be preceded by a fontlist tag component.

A compound string can be used for the label of a widget by specifying a *fontlist* for the widget's font resource. A fontlist is a set of (font, tag) pairs. The fontlist tags indicate which font in the fontlist to use for each text string component.

To familiarize yourself with the features of Sun WorkShop Visual's compound string editor, use the following step-by-step example while running Sun WorkShop Visual at your workstation.

- **Select "New" from the File Menu.**

The object of the exercise is to reproduce the masthead of the London Guardian on a Label widget, shown in Figure 5-15.

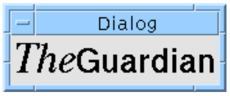


FIGURE 5-15 Final Appearance of the Text String

Creating a Complex Font Object

So far, you have learned how to select a font and apply it to a widget. You have also created a simple font object that corresponds to a single font. *Complex font objects* let you produce more complex visual effects. A complex font object corresponds to a list of fonts; different parts of a string can then be displayed using different fonts from the list. The step-by-step example in this section requires a complex font object.

Create a widget that uses a label and give it a font object with more than one font in its list.

1. **Create a widget hierarchy with a Label as the child of a Form.**
2. **Double-click on the Label widget to display the resource panel.**
3. **Click on the "Font" resource button to display the font selector.**
4. **Use the pulldown menus to select a 24-point medium italic Times font.**
5. **In the Font object "name" field, type:** `masthead`
6. **In the "Fontlist tag" field to the right of the "Font object" field, type:** `italic`
7. **Click on "Bind".**

This creates a font object called *masthead* with one font in its list. The font is tagged *italic*.

Note – Tag names are arbitrary. However, several pre-defined tags are listed by the “Default” button and the “UIL” pulldown menu. The “Default” and “UIL” buttons are above the “Fontlist tag” field. Selecting the “Default” option produces the tag *XmFONTLIST_DEFAULT_TAG*. The UIL menu lists tags that can be used in UIL (Motif’s User Interface Language). If you are a UIL user, note that most UIL compilers produce a “severe internal error” if you use a tag that is not on this list. If you are not a UIL user, ignore the UIL options.

Next, add another font to the font object. The second font has a different tag.

8. **Select a 24-point bold regular Helvetica font.**

9. **In the “Fontlist tag” field, type:** `bold`

   Do not change the *masthead* font object name.

10. **Click on “Bind”.**

    This adds the bold font to the font object list and tags it “bold”. You can see samples of the two fonts in the “Sample” field by selecting the different tags in the “Fontlist tag” list.

    You now have a font object that can be used to display a string using two different fonts. Apply it to the Label widget:

11. **Click on “Apply” in the font selector.**

12. **Click on “Apply” in the Label resource panel.**

    The text in the Label is displayed in the italic Times font because it is the first in the list. To display a text string that uses both fonts, you must create a compound string.

Creating A Compound String

When you type text into the “Label” text field of the resource panel, Sun WorkShop Visual creates a compound string which only contains text string components and separator components, which correspond to the newlines. To create a compound string that includes fontlist tags, you must use the compound string editor.

1. **Click on the “Label” button in the resource panel.**

   This displays the compound string editor, shown in Figure 5-16.

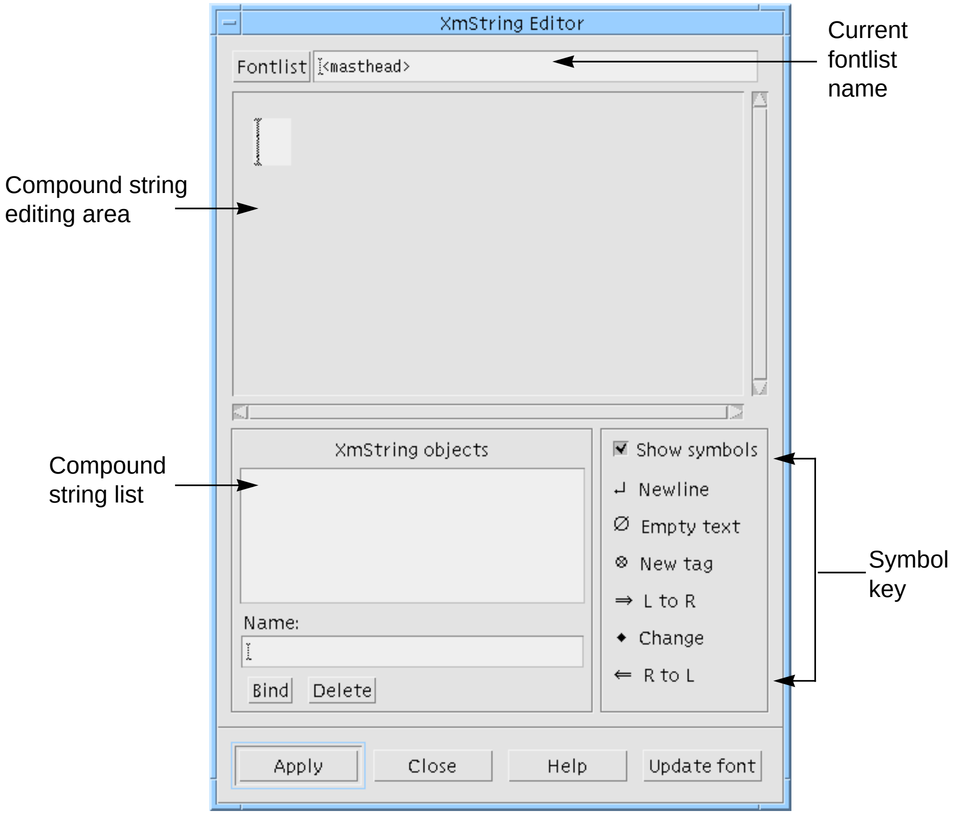


FIGURE 5-16 Compound String Editor

The compound string editor includes the current fontlist name, an editing area and a list of existing compound string objects. The fontlist name is currently “masthead”, to correspond with the name of the font object used in the widget.

As you enter text in the compound string, it appears in the editing area. The fontlist named at the top of the screen is used to display the text. Other components, such as empty text components and directional indicators, appear as symbols. You can turn off the symbol display using the “Show symbols” toggle to see how the text will look in the widget.

If the “Show symbols” toggle is not on:

2. **Click on the “Show symbols” toggle.**

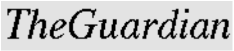
To create the compound string, start by entering the text:

3. **Click on the I-beam cursor in the editing area.**

This makes the cursor blink, indicating that the editor is ready for text entry.

4. **Type the following string with no space between the words: `TheGuardian`**

   The text is displayed using the first font in the fontlist, which is the italic Times font, as shown in Figure 5-17.



TheGuardian

FIGURE 5-17 Initial Text for the Compound String

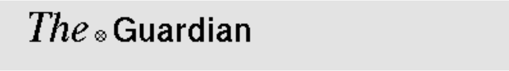
To make different parts of the text display in different fonts, you must insert fontlist tags into the compound string.

5. **Position the pointer between “The” and “Guardian”, then press and hold mouse button 3.**

   A popup menu appears. Selecting an item from this menu inserts a component of the corresponding type into the string. Selecting “Delete” deletes the current component. To display the word “Guardian” in the bold Helvetica font, you must insert the appropriate fontlist tag (“bold”) at the pointer.

6. **Pull right for the Fontlist tags Menu and select “bold”.**

   This inserts the fontlist tag symbol and changes the display as shown in Figure 5-18.



The ⊗ **Guardian**

FIGURE 5-18 Compound String with Fontlist Tag

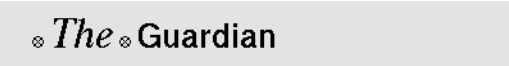
If you accidentally insert the tag in the wrong position, you can pick it up and move it using mouse button 1.

While the word “The” is now correctly displayed in the italic font, this is only because Motif uses the first font in the fontlist by default. To make the compound string complete you should insert the *italic* tag at the beginning of the string.

7. **Position the pointer before “The”, then press and hold down mouse button 3.**

8. **Pull right and select “italic”.**

   This inserts a second fontlist tag symbol, as shown in Figure 5-19.



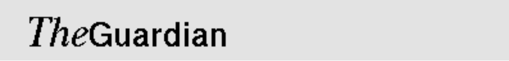
⊗ *The* ⊗ **Guardian**

FIGURE 5-19 Compound String with Fontlist Tags

The compound string is now complete. To see it as it will appear on the label:

9. **Click on the “Show symbols” toggle to turn off the symbols.**

   This redraws the string without the fontlist tag symbols, as shown in Figure 5-20.



*The*Guardian

FIGURE 5-20 Compound String Without Symbols

Creating a Compound String Object

Now you can create a compound string object and bind it to the string.

1. **In the field labelled “Name” underneath the list of XmString objects, type: `guardian`**
2. **Click on “Bind”.**

   The name of the object appears in the *XmString* object list. Finally, you can apply the compound string object to the Label.

3. **Click on “Apply” in the compound string editor.**

   This applies the compound string object to the Label resource panel.

4. **Click on “Apply” in the Label resource panel.**

   The text in the Label now displays the compound string in the selected fonts.

Direction Indicators

So far, this exercise has demonstrated two of the components in a compound string: the text and fontlist tags. The other two components are the newlines used as delimiters and direction indicators.

A newline causes a line break in the text. You can either press <*Return*> while entering text, or insert a newline using the pull-down menu. Before experimenting with inserting, moving and deleting newlines, make sure the “Show symbols” toggle is on so you can see what you are doing.

The direction indicators let you create text in either a left-to-right or a right-to-left direction. Although the default direction is from left to right, you can specify any portion of the string to be drawn from right to left.

If the “Show symbols” toggle is not on:

1. **Click on the "Show symbols" toggle to display the symbols.**
2. **Position the pointer between "Guar" and "dian", then press and hold mouse button 3.**
3. **Pull down and select "Right to left".**

   The compound string changes as shown in Figure 5-21.

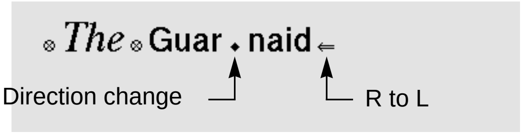


FIGURE 5-21 Compound String with Direction Change

The small diamond represents a change of direction from left-to-right to right-to-left. The arrow is the direction indicator, which appears at the beginning of the affected text. Since this is a right-to-left segment, the beginning of the text is on the right, not on the left. A change to left-to-right is represented by a direction indicator arrow pointing to the right at the beginning (the left end) of the new text.

Inserting newlines and fontlist tags into a right-to-left segment may seem to produce the opposite effects from what you expect if your normal reading direction is from left to right.

Remember that symbols affect the text that follows them, which means the text to the left in a right-to-left segment.

Finally, bind the compound string object *guardian* to the new string.

4. **Click on "Bind".**

   The Label text changes as shown in Figure 5-22.

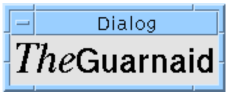


FIGURE 5-22 Final Appearance of the Text String

String objects can be shared between files by making them "global". This is controlled by setting the appropriate toggle when you generate code. See "Global Object Functions" on page 219 for more information.

Updating Changes to the Font

If you create a new font object, or decide to use a different one while using the compound string editor, you can update the "Font" field in the widget's resource dialog by pressing the "Update font" button in the Compound String Editor. This ensures that the widget is using the same font object as the editor. This is not necessary if you have changed the contents of the font object - only if you want to use a different one.

Callback and Prelude Editing

Sun WorkShop Visual provides a means of editing callbacks and code preludes. This facility is available once you have specified a callback and generated a stubs file or, in the case of code prelude editing, you have defined a code prelude and generated the primary source file.

- See "Adding Callback Functionality" on page 226 for details on adding a callback.
- See "Setting up the Stubs File" on page 213 for details on generating a stubs file.
- See "Customizing the Generated Files: Preludes" on page 239 for details on code preludes.
- See "Setting up the Primary Source File" on page 211 for details on generating the primary source file.

You must first select the widget which has been given the callback or prelude. For callbacks, the editing capabilities are available from the Callbacks dialog. For code preludes they are available from the Widget menu. See "Designating a Callback" on page 173 for details on the Callbacks dialog.

When you ask to edit a callback or prelude you are given a text editor running inside an Xterm window. The editor used is determined by the *editor* application default which is described in Appendix D, "Application Defaults".

If you have not already generated a stubs file or primary source file, you are prompted to do so.

More about callback editing is provided in "Adding Callback Functionality" on page 226. More about code prelude editing is provided in "Code Preludes" on page 242.

CHAPTER 6

Activating the Interface: Adding Your Own Code

Introduction

The aim of Sun WorkShop Visual is to let you develop as much of your application as possible without writing code. You still need to write code, however, to make your application work as you intend and to link it to the user interface. You must also write code to control the behavior of the user interface. The following Sun WorkShop Visual features make some of this easier:

- Callbacks. You can name a callback function or method for individual widgets and generate a skeletal procedure for it.
- Links. Some commonly used callback events are known to Sun WorkShop Visual, enabling you to add them graphically and immediately view the effect.
- Drag and Drop. You can set up a widget to receive data via the Motif drag and drop mechanism.
- Translations and Actions. You can change the way individual widgets respond to events.
- Xt Procedures. Sun WorkShop Visual provides a quick and easy means of adding Xt Work, Input, Timeout, Action and Language procedures and Event handlers to your design.

This chapter uses the tutorial example built in the preceding chapters to demonstrate how to add a callback and how to add links to your design. The above topics are also examined separately in some detail.

Callbacks

A *callback list is* a list of one or more *callback functions* designated to be triggered by *user actions* in your application. User actions include mouse button presses, keyboard selections and movements of the pointer. By setting up a *callback*, you can instruct your interface to call the functions on the callback list whenever a certain user action occurs within a widget. The callbacks dialog is shown in Figure 6-1.

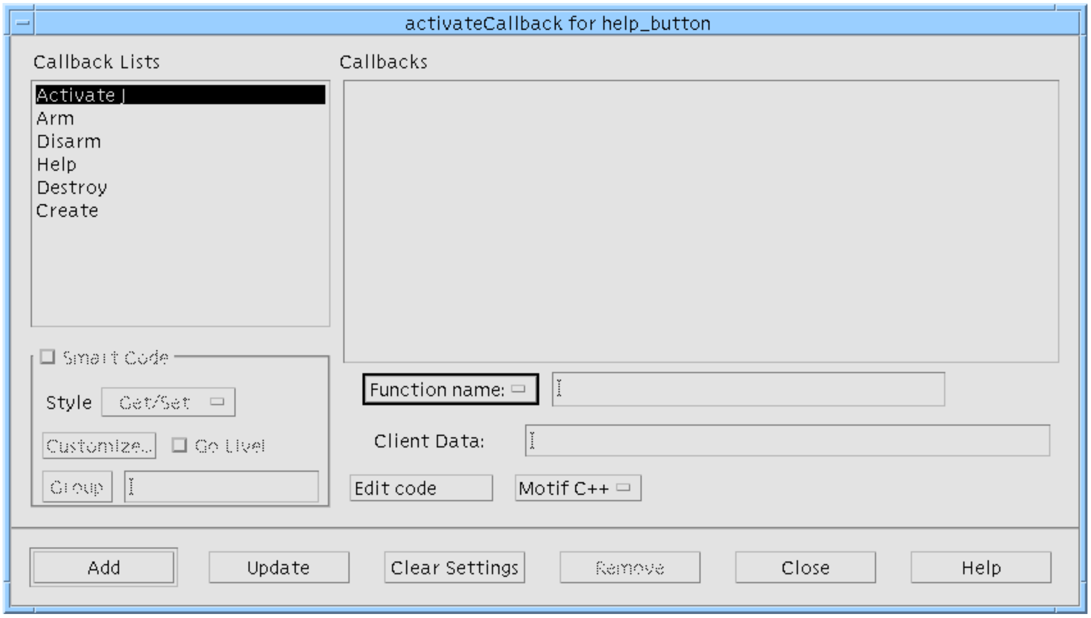


FIGURE 6-1 Callbacks dialog

An “M” or “C” displayed to the right of a callback indicates that a method or callback respectively has been declared.

Callbacks which apply to Java code are listed with the letter ‘J’ after them, as shown in Figure 6-1. Callback functions are not generated into the Java code, so you should use callback methods if you wish to use them in your Java application. For more information on using Sun WorkShop Visual to generate Java code, see Chapter 10, “Designing for Java”.

An asterisk (*) next to a callback indicates that the callback is not supported by Microsoft Windows.

The area in the lower left of the Callbacks dialog allows you to set up a Smart Code callback. These are callbacks which give you toolkit independence by “wrapping” specified widgets in an extra layer of code. Smart Code is most useful when you are creating a thick client, or a thin client and server, from your design. See Chapter 16, “Get/Set Smart Code”, starting on page 485 for more information on Smart Code. For this tutorial, you will not be using Smart Code. However, there are a number of tutorials which do use Smart Code in later chapters.

Designating a Callback

In the following steps you will designate the simplest example of a callback by using the design from the tutorial of the preceding chapters. Clicking on the “Exit” button (*exit_button*) will trigger a callback list with just one function, *quit(),* which terminates the program.

The “Callbacks” dialog lets you associate lists of functions with user actions. You can associate *quit()* with *exit_button* now, even though *quit()* has not yet been written.

quit() and other callback routines are written in C or C++ and linked in with the code generated by Sun WorkShop Visual. You will write your *quit()* routine in “Adding Callback Functionality” on page 226. The topic of writing callbacks is discussed in greater depth in “Callback Functions” on page 178.

1. **Open your saved tutorial design.**
2. **Click on the “Exit” button (exit_button).**
3. **Click on the “Callbacks” toolbar button or select “Callbacks” from the Widget menu.**

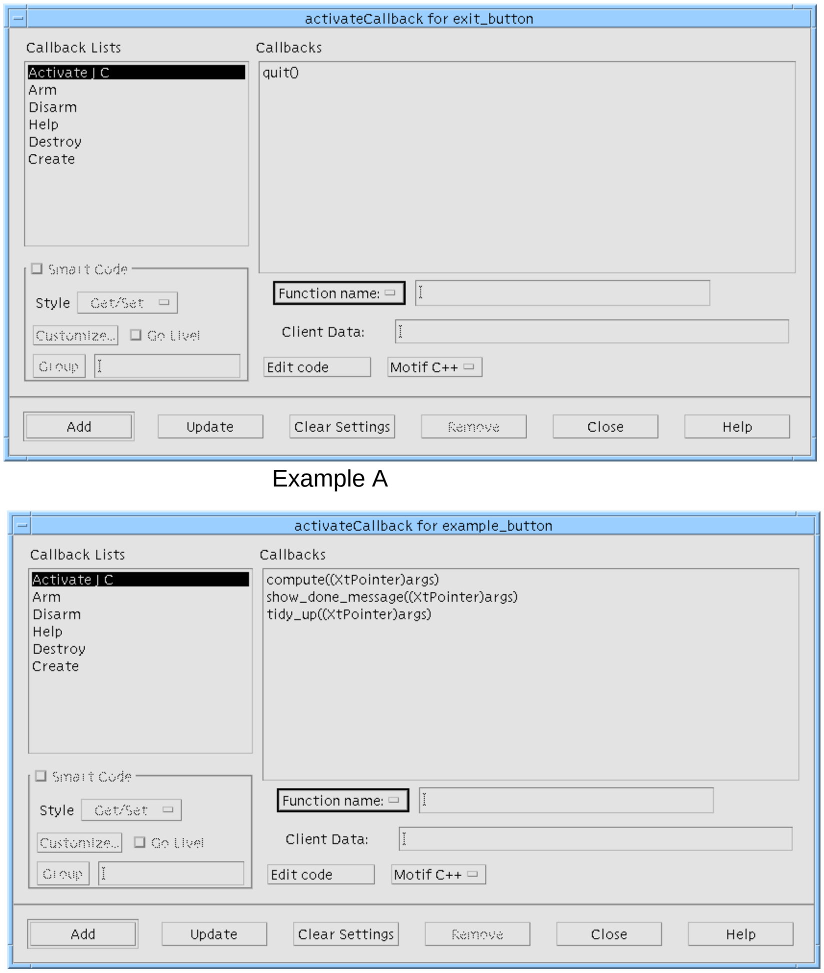
   You are going to associate *quit()* with the “Activate” callback. Activating means that the user presses and then releases a mouse button while the pointer is located inside the widget. The user can also activate with the <*Return*> key, or other keys as described in the *Motif User Guide*.

   When a callback in the “Callbacks” dialog is selected, the list of callback functions you have associated with it are displayed. To add an Activate callback:

4. **Click on “Activate” in the list of Callback types.**

   This displays, in the list of callback routines, both those callbacks local to the widget and those *inherited* by it. Inherited callbacks are explained in “Inherited Callbacks” on page 174.

You may only change callbacks which have not been inherited. Figure 6-2 shows two typical examples. Example A of the figure shows the callback list for the “Exit” button in the tutorial interface; Example B shows a slightly more complex list.



```
quit()
```

Example A

```
compute((XtPointer)args)
show_done_message((XtPointer)args)
tidy_up((XtPointer)args)
```

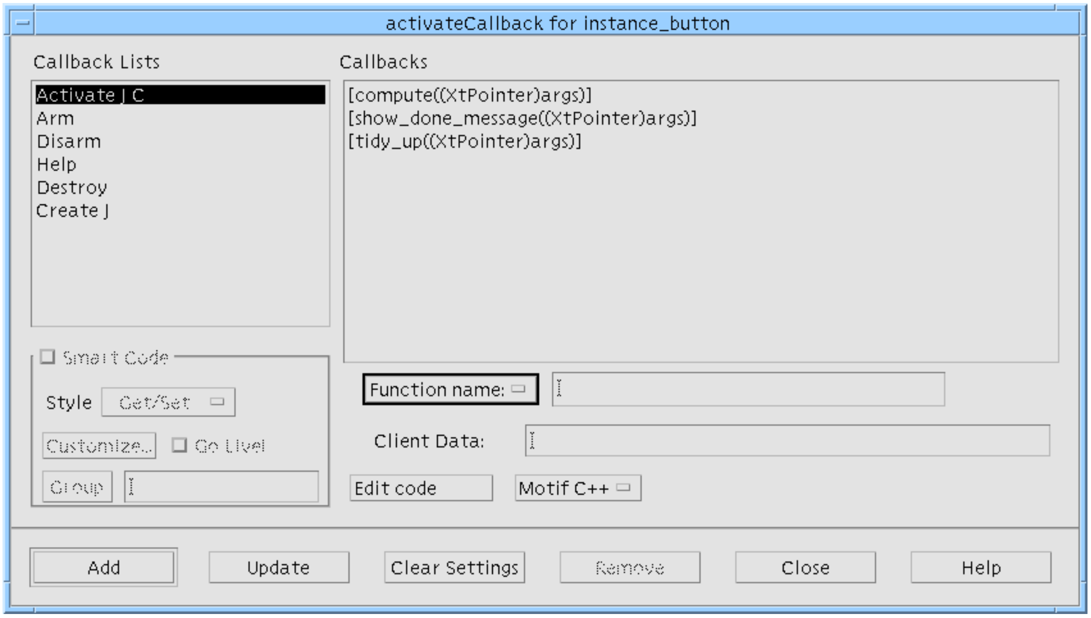
Example B

FIGURE 6-2 Callback Text box and Two Examples of Syntax

Inherited Callbacks

Widgets which are instances of definitions can inherit callbacks from the corresponding widget in the definition.

Inherited callbacks are shown enclosed in square brackets ([]), as shown in Figure 6-3.



```
[compute((XtPointer)args)]
[show_done_message((XtPointer)args)]
[tidy_up((XtPointer)args)]
```

FIGURE 6-3 Inherited Callbacks

Callback Syntax

In general, the syntax for each function call in the callback list is the same as C syntax. You do not, however, type them in as C code. Function brackets () are not required as these are added automatically.

Note – If you add function brackets or parameters to the name of the callback function or method these will be treated as part of the name - they will not be recognized as syntactically separate.

When we specify the name of the callback, we must also choose which language we are using.

1. **If it is not already selected, choose "Function name" from the option menu.**
2. **Click in the "Function name" text box.**
3. **Type: `quit`**

4. **Click on “Add”.**

   Pressing Return has the same effect.

5. **Close the dialog.**

6. **Save your design.**

Order of Execution

While the callback list looks like C code, it has no logical flow. This means that you can neither use C’s logical operators such as *if...else* and *while*, nor can you rely on your callbacks being executed in any particular order. All routines in your list will be executed whenever the specified event occurs but not necessarily in the order you type them. If the execution sequence is important, you can write a single callback function which contains subroutine calls in the order you want.

Client Data

The “Client Data” text box allows you to specify data to be passed in to the callback. It is better practice to use this mechanism when a callback needs to use some data than to use a global variable. Enter the string you wish to appear as the parameter here. You may also add a type cast in the usual C/C++ syntax. You do not, however, need to type the function parameter brackets () as these will be added for you automatically. See “Callback Function Parameters” on page 178 for more details about the parameters passed to callback functions and an example.

Note – You can only add client data to function callbacks, not to method callbacks. See “Callback Methods” on page 259 for more details about this.

Methods

If the selected widget is enclosed in a C++ class and “Method name:” is selected from the pulldown menu to the left of the name text field, a “Methods...” button is shown to the right of the name text box. With “Function name:” selected, this button is not shown. When pressed, the “Methods...” button displays a list of callback methods already defined for the selected widget’s enclosing class.

Selecting one of these and pressing “OK” will place the selected item in the “Method name” text field. See “C++ Classes” on page 254 for details about the way in which widgets are enclosed in classes.

Edit Code

Pressing this button allows you to edit your stubs file (the generated file containing the specified callbacks) without leaving Sun WorkShop Visual. This is dealt with in more detail in “Adding Callback Functionality” on page 226. See also “Setting up Callback and Prelude Editing” on page 701 for details on how to set up the editing feature so that you can use the editor of your choice.

Flavor Option Menu

The option menu next to the “Edit Code” button contains the possible code generation “flavors”. The options are:

- Motif C
- Motif CPP
- Motif XP
- Microsoft Windows MFC

Note – The last two options are only shown in Microsoft Windows mode. See “The Flavor Menu” on page 362 for information on these.

The Flavor Option Menu works in conjunction with the “Edit Code” button. When you edit your stubs file, you must specify which language you are using. Sun WorkShop Visual will try to work this out for you and set the menu accordingly when you invoke the dialog. Sometimes this is not possible if, for example, you are working with two languages in the same design. You should, therefore, always check that the appropriate option is chosen from this menu.

Update

Use the “Update” button to change the settings of an existing callback.

If you wish to change a non-Smart Code callback to a Smart Code callback you can only do so if the callback is used in just one place. This is because the same callback cannot be both at the same time. If you select a non-Smart Code callback that is being used elsewhere, the Smart Code toggle remains insensitive.

If an existing callback has been changed into a Smart Code callback and you have already generated a stubs file, you should go to the file and remove or rename the non-Smart Code callback stub so that Sun WorkShop Visual will generate the new Smart Code for you

Clear Settings

Using the “Clear Settings” button is most useful if you have one or more Smart Code callbacks and you wish to add a new callback routine without accidentally picking up the Smart Code settings of the previously selected callback. “Clear Settings” deselects all callbacks and any other non-default settings.

Remove

The “Remove” button removes the currently selected callback. Be careful when using this button as the operation cannot be undone.

The tutorial example continues in “Links” on page 183. The chapter now continues with a more thorough explanation of callbacks and how they may be used.

Callback Functions

“Creation Procedures” on page 232 describes how Sun WorkShop Visual creates the widgets in your application and sets their initial resource values. However, it is the callback functions and translations that make the application work. See “Translations and Actions” on page 190 for more information on translations.

Most callback functions have a similar structure. A typical callback function does some or all of the following:

- Extracts information from widgets, such as the text in a Text widget or the state of a ToggleButton
- Uses this information as parameters for calls to application functions
- Uses the results of these functions to change widget attributes. These attributes include not only values (such as the text in a Text widget) but also sensitivity (responsiveness to user input), visibility and even existence

Callback Function Parameters

A callback function receives three parameters:

- The widget from which the callback was invoked
- The call data
- The client data

The call data is a pointer to a data structure defined by the widget developer. Call data structures are described in your Motif documentation or documentation supplied by the developer of the widget toolkit. See “Books on X and Motif” on page 886 for details of some useful Motif documentation.

The client data is a pointer that you can use to pass the address of any variable or structure. When you register a callback, you can specify the value for the client data parameter that is passed to the callback function.

In Sun WorkShop Visual, the client data is specified in the callback dialog as a single optional parameter of the callback function. See “Client Data” on page 176 for details on how to do this. This could be a pointer to a structure, which can be defined and initialized in a suitable prelude. For example, a typical prelude might be:

```
/* Pre-manage prelude for main dialog Shell */
/* Define and initialize client data for the rungrep callback */
static rcdata_t rcdata = {
    &hitstring,
    &errorshell,
    &errorform,
    &errortext,
    &mainshell
};
/* End of Shell pre-manage prelude */
```

The callback is specified as:

```
rungrep((XtPointer)&rcdata)
```

To enter this in the Callbacks dialog, the function name `rungrep` is typed into the text field labelled “Function name” and the parameter, including the cast, is typed into the “Client data” text field: `(XtPointer)&rcdata`.

The declaration of the structure *rcdata_t* would normally be in a header file that would be included in the generated code (by adding “*#include ...*” as the module prelude) and in the callback function module. The callback function can then cast the client data to *(rcdata_t *)* and so access the data.

Note that the structure *rcdata* is defined to contain pointers to the widget variables, rather than the values of the variables themselves. This lets *rcdata* be initialized before the widgets are created. You can also define a structure into which the values of the widget variables are copied. However, this cannot be initialized until all the widgets have been created, which can be tricky.

Callbacks in C++

Ideally it would be desirable to add class member functions to widgets as callback functions. Unfortunately this is not possible because callback functions are called by a C library and they cannot provide the instance context (the *this* pointer) required by a class member function. Sun WorkShop Visual provides an automatic way of calling a class member function from a callback. These are called callback methods and are discussed in "Callback Methods" on page 259.

Accessing Widgets in Callbacks

All callbacks are passed the address of the widget to which they belong. This is a variable of type *Widget*. In the Sun WorkShop Visual generated code, the variable name of the widget is used for the name of this pointer. If you want a callback function to access widgets in your design other than the widget to which the callback belongs, you have the following choices:

Client Data Structure

Pass the other widgets as part of the client data structure. See "Callback Function Parameters" on page 178 for a description of this structure.

Global Widget Variables

The simplest technique is to have Sun WorkShop Visual define the widget variables as global. You can then access them from a callback function by declaring them as *extern* in the callback function module. You could include Sun WorkShop Visual's generated "Externs" file in the stubs file in order to do this.

Sun WorkShop Visual declares named widgets as global by default. You can change this behavior by setting the Storage Class of the widget in the Core resource panel.

The strength of the global variable approach is its simplicity. However, having many global variables does nothing for the structure of your program and you must pay attention to naming conventions to ensure meaningful names and avoid duplicates.

Inclusion of Generated Code

You can reduce the need for global variables by including the primary module in the callback function module, using *#include*. The primary module should be generated without includes of the X and Motif header files.

If you do this, Sun WorkShop Visual still declares named widgets as global. You may want to change their storage class to static, which makes them local to the callback function module.

This technique works well where a callback function needs access to widgets that are all or mostly within a single design. In more complex situations, you can add accessor functions to the callback function module. A callback function that needs to manipulate a widget which is local to another callback module can do so via the accessor functions.

You can also access widgets using the X toolkit convenience function `XtNameToWidget()`. Pass the widget name to this function and a pointer to the widget is returned. See your X toolkit documentation for more information.

Manipulating Widgets

"Accessing Widgets in Callbacks" on page 180 gives you some ways to access the widgets in your design. Once you have a widget, there are many ways you can manipulate it. This section outlines a few of them. It is not a detailed description, but is only intended to point you to the appropriate functions and their documentation.

Toolkit Convenience Functions

The Motif toolkit provides a large number of convenience functions for getting and setting attributes of some widgets. These are all named after the widget class that they affect, such as *XmTextSetString()*, *XmTextGetString()*, *XmToggleButtonGetState()*. These are documented in the *Motif Programmer's Reference*.

Convenience functions are the first place to look. They are the easiest to use and are likely to be efficient.

One point to note is that convenience functions take a *Widget* parameter and expect this to be a pointer to a widget of the appropriate class. If the widget is of the wrong class, they commonly core dump. There are also both widget and gadget versions of some of the convenience functions and you may get a core dump if you use the wrong one.

Setting and Getting Resources

If there is no convenience function, you may have to get or set one or more of the resources of the widget directly using *XtGetValues()* or *XtSetValues()*. This is fundamental to widget programming and any book on Xt or Motif should cover it adequately.

Not all resources can be set after a widget has been created. The *Motif Programmer's Reference* documents the access controls on each resource of every widget class.

Enabling and Disabling Widgets

To disable a widget (that is, to make it insensitive to user input), or enable it again, use *XtSetSensitive()*. You should not set the resource *XmNsensitive* directly.

When a widget becomes insensitive, so too do all its descendants. Insensitive widgets are usually grayed out.

If you make a Text or TextField widget insensitive, the user cannot use key input to pan and scroll the text and so has only a limited view. It may be better to set the resource *XmNeditable* to *False*.

Showing and Hiding Widgets

There are two ways to make a widget appear or disappear: managing and mapping.

If a widget is unmanaged, its parent does not reserve any space for it and it is not visible on the screen. A widget is unmanaged using *XtUnmanageChild()* or *XtUnmanageChildren()* and managed using *XtManageChild()* or *XtManageChildren()*. Sun WorkShop Visual generates code to manage widgets after they have been created, but the Managed toggle in the Core resource panel changes this.

If a widget is managed but not mapped, its parent reserves space for it. However, it is still not visible; there is a blank hole. Widgets are normally mapped automatically when they become managed. This is controlled by the resource *XmNmappedWhenManaged* which can be found on the "Settings" page of the Core Resource panel.

Mapping and unmapping is commonly used to change the visibility of widgets within a dialog without causing its layout to change. Managing and unmanaging cause layout changes.

You can make a complete dialog appear or disappear by managing or unmanaging the child of the Dialog Shell. If the dialog uses a Top level Shell, use *XtPopup()* and *XtPopdown()* on the Shell instead.

Note – “Links” on page 183 describes an automatic and dynamic means of showing and hiding widgets using Sun WorkShop Visual‘s built-in links facility.

Creating and Destroying Widgets

Sun WorkShop Visual generates code to create the widgets for your dialogs. The default *main()* program calls all the creation functions at start-up time. Since widget creation is relatively expensive, this may cause an unacceptable delay. It is common practice to defer creation of a dialog until the first time it is popped up. A static Boolean flag in the callback function that performs the popup can be used to determine if the dialog has already been created.

As well as generating code to create complete dialogs, Sun WorkShop Visual can generate creation functions for dialog fragments, as described in “Children Only Place Holders” on page 265. You can call these from a callback function, so as, for example, to create another instance of some reusable component.

To destroy a widget (and all its children), use *XtDestroyWidget()*. It is inefficient to destroy a widget and then recreate it; you should unmanage it, then manage it again when it is needed.

Links

Sun WorkShop Visual has predefined callback procedures called *links*. There are six links available:

- *Show* – makes a widget and its children appear on the screen
- *Hide* – makes a widget disappear. The widget is not destroyed, just hidden
- *Manage* – manages a widget which has already been created
- *Unmanage* – unmanages a widget
- *Enable* – enables a button or command
- *Disable* – disables (grays out) a button or command

Distinction Between Links and Callbacks

Only PushButtons, ArrowButtons and CascadeButtons can be the *source* of a link. All links are triggered by an “Activate” event. A link can show, hide, manage, unmanage, enable, or disable any widget in the design. One button can have multiple links.

Links are callbacks which Sun WorkShop Visual sets up for you. Unlike callbacks, however, links work in the dynamic display and can therefore be used for prototyping window behavior. When you generate code, you can either include links, which work exactly as they do in the dynamic display, or substitute more complex callbacks for the simple links.

Restrictions on Adding Links

Links can only be added if at least one of the following criteria is met. If none are met, the “Add” button is disabled and no links can be made. The requirements are:

- The *target widget* of a link - that is, the widget to be shown, hidden, managed, unmanaged, enabled, or disabled - must have an explicit variable name. The *source* widget does not have to be explicitly named.
- If the target widget is a Shell, its immediate child must also be explicitly named.
- The widgets on either end of a link must not be designated as *static* or *local* variables. See “Changing Declaration Scope” on page 266 for a discussion of static and local widget variables.
- The target widget must not be declared “children only”. See “Children Only Place Holders” on page 265 if you are not sure what this means.
- If the target widget is part of an instance of a definition, the root widget of the instance must also be named. See “Instances” on page 273 for more information on instances.
- If the target widget is part of an instance of a definition which is a C++ class, it must be declared “Public”. See “C++ Classes” on page 254 for more information on C++ classes and member access.
- If you are generating Java code, the source widget must be contained within a class and not be abstract. The term ‘abstract’ refers to the ‘extra’ widgets which appear in the hierarchy when composite widgets are selected - such as the ScrolledText in a FileSelectionBox. For more information on using Sun WorkShop Visual for Java code generation, see Chapter 10, “Designing for Java”.

Note – If the variable name of a target widget changes, any links defined to that widget are no longer effective.

Setting Links in the Tutorial

You are now going to set a common configuration of links to display the help screen you have just built and make it disappear again at the proper time. To do this you will:

- Set a “Show” link on the “About This Layout” button in the Help Menu
- Set a “Hide” link on the “OK” PushButton in the help screen

The “OK” PushButton is currently visible in the construction area and so begin by setting the “Hide” link on this PushButton.

1. **Select the PushButton.**
2. **Double-click in the “Variable name” field.**
3. **Type: `ok_button` and press <Return> to register the new name.**
4. **Pull down the Widget Menu and select “Edit links”.**

This displays the panel shown in Figure 6-4.

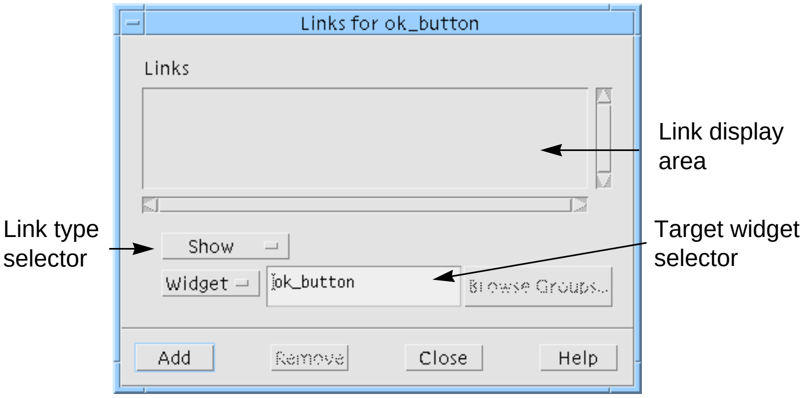


FIGURE 6-4 Default Links Panel

The target of the “Hide” link should be the widget *help_window* so that when the “OK” button is activated, the entire help screen disappears.

To select the target widget:

5. **Select the Shell `help_window` in the design hierarchy.**

The name of the Shell, *help_window*, appears in the “Widget” field of the Links panel. However, the “Add” command is still disabled. This is because you have not yet named the DialogTemplate which is the immediate child of the Shell. As discussed above, the child of a Shell must be named explicitly before you can set a link to the Shell.

You can leave the Links panel open while you name the DialogTemplate:

6. **Select the DialogTemplate.**

7. **Double-click in the “Variable name” field and type: `dialog_2`**

8. **Reselect the Shell `help_window`.**

   The Shell is now a valid target widget and so “Add” is enabled.

   Now select the type of link:

9. **Select “Hide” from option menu of link types.**

10. **Click on “Add”.**

    The new link appears in the link display area, as shown in Figure 6-5.

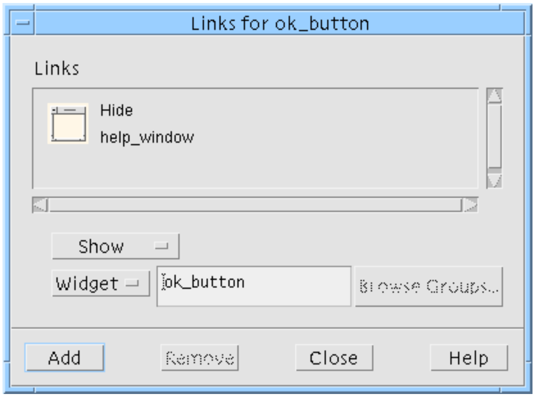


FIGURE 6-5 Links Panel with New “Hide” Link

11. **Click on “Close”.**

    To demonstrate the new link:

12. **Click on the “OK” button in the dynamic display.**

    The help screen vanishes. You can restore it by resetting the Shell.

    You can also set up a “Show” link to display the help screen when a button is pressed in the main window. To do this:

13. **Click on the *MyFirstShell* icon in the window holding area.**

    The hierarchy for the main window is displayed in the construction area.

    Set the new link on the PushButton in the Help Menu:

14. **Select the *help_button* widget, the PushButton child of the second Menu.**

    The Links panel, unlike resource panels and the Layout Editor, does not automatically start adding links to the currently selected PushButton. To edit links for the currently selected button, you must:

15. **Pull down the Widget Menu and select “Edit links”.**

    The Links panel now displays the name and the links (none, so far) of the current PushButton. Select the target widget, which is the Shell for the help screen:

16. **Click on the *help_window* icon in the window holding area.**

    In the Links panel:

17. **Select the “Show” link type.**

18. **Click on “Add”.**

    The new link appears in the link display area.

    To demonstrate the behavior of these two links:

19. **Click on the *MyFirstShell* icon in the window holding area.**

20. **Pull down the Help Menu in the dynamic display and select “About This Layout”.**

    The Show link on this pushbutton makes the help screen appear.

21. **Click on the “OK” button in the dynamic display of the help screen.**

    The Hide link on this pushbutton makes the help screen disappear. You can repeat the previous two steps as many times as you want.

22. **Save your design.**

Removing Links

To remove a link:

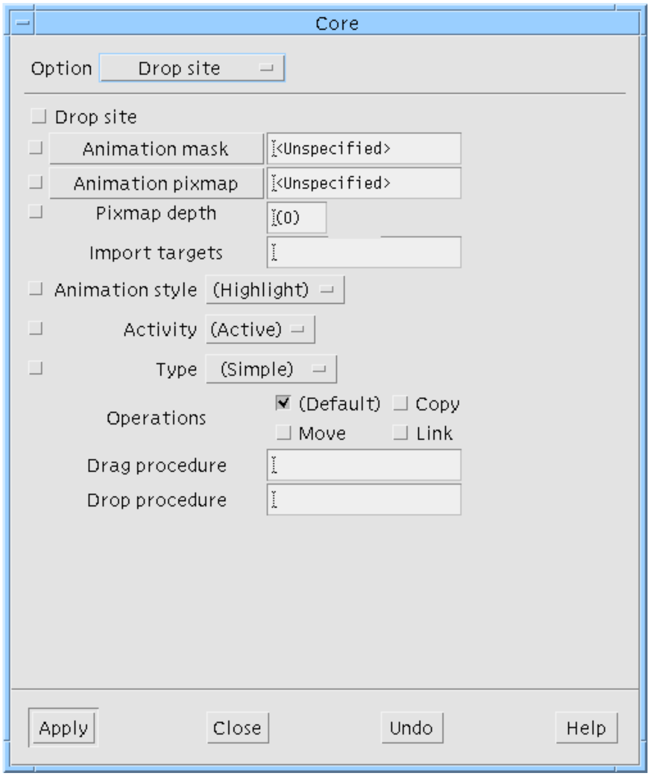
- **Select the link’s icon in the link display area and click on “Remove”.**

The tutorial continues in Chapter 7, “Generating Code”. The remainder of this chapter looks at other ways of adding functionality to your application using Sun WorkShop Visual.

Drag and Drop

Motif 1.2 provides a sophisticated drag and drop mechanism that lets applications communicate data via the X selection mechanism. Sun WorkShop Visual provides some simple support to let you specify drop sites in your application. Because the initialization of a drag is a dynamic function that would normally be done from within a callback or action function, Sun WorkShop Visual does not provide any explicit support.

A drop site is a widget that is prepared to receive certain types of data from the transfer mechanism. Sun WorkShop Visual provides its support through the Drop site page of the Core resource panel.



Core

Option: Drop site

□ Drop site

□ Animation mask <Unspecified>

□ Animation pixmap <Unspecified>

□ Pixmap depth: (0)

Import targets:

□ Animation style: (Highlight)

□ Activity: (Active)

□ Type: (Simple)

Operations: ☑ (Default) □ Copy □ Move □ Link

Drag procedure:

Drop procedure:

FIGURE 6-6 The Drop Site Page

To designate a widget as a drop site, simply set the “Drop site” toggle on and specify the import targets and drop procedure. The “Import targets” field is a list of strings that are converted into atoms to designate types that can be handled by the drop procedure. The list is specified as strings separated by commas or spaces.

By default Motif makes Label (and derived) widgets start a drag operation to transfer the *labelString* or *labelPixmap* if Button 2 is pressed over them. Sun WorkShop Visual takes advantage of this by adding a drop procedure to the drop site widget that imports these types if specified in the import targets. The following tutorial lets you see how the drop site operates.

1. **Create a dialog containing an Application Shell with a RowColumn containing two Push Buttons.**
2. **Name the widgets:** *shell*, *rowcolumn*, *MyButton1* **and** *MyButton2* **respectively.**
3. **Pop up the Drop site page of the core resources for** *MyButton1*.
4. **Set the drop site toggle on and set animation style to “shadow in”.**
5. **In the “Import targets” field, type:** `COMPOUND_TEXT`
6. **In the “Drop procedure” field, type:** `drop_button1`
7. **Apply your changes and close the dialog.**

   The drop and drag procedure fields specify the names of functions to be called to handle the drop and dynamic drags respectively.

8. **Try dragging the text from any label (press mouse button 2 and drag) across** *MyButton1* **in the dynamic display window.**

   The button is “shadowed in” i.e looks selected, to indicate that it is a valid drop site for the target being dragged.

9. **Release the mouse button to drop the text into the widget.**

   The drop procedure provided by Sun WorkShop Visual simply copies the label into the widget.

10. **Select** *MyButton2* **and repeat Step 4.**
11. **In the “Import targets” field, type:** `PIXMAP`
12. **In the “Drop procedure” field, type:** `drop_button2`
13. **Try dragging a pixmap from the tool bar across MyButton2 in the dynamic display window.**

    For further examples of using drop sites and for information on starting drags, refer to the Motif documentation.

Code is generated for C and C++, with a call to *XmDropSiteRegister()* being generated for widgets that are not normally drop sites. Text widgets are drop sites by default, which can import *COMPOUND_TEXT*. This can be disabled by setting the drop site toggle off, or modified by simply changing the appropriate resources.

You must write the drop procedures to handle the transfers. They are simply declared as external functions in the generated code.

Translations and Actions

Widgets have behavior. For example, when a user presses mouse button 1 over a PushButton, it highlights. When the user releases the mouse button, the PushButton's appearance reverts to normal and the functions on the Activate callback list are invoked.

This behavior is not hard-wired into the PushButton widget. Instead, it is determined by the widget's *translation table*, which maps events to the actions to be taken in response to the events. When a widget is created, its translation table is initialized to contain a default set of entries. For example, the PushButton widget's default translation table includes these entries:

```
<Btn1Down>:Arm()
<Btn1Up>:Activate() Disarm()
```

To the left of the colon is an event specification; to the right are the names of the actions that the widget performs in response. A second table, the action table, is used to map the action name to the address of a function that performs it.

For example, the PushButton's default action table includes:

```
“Arm”,Arm
“Activate”,Activate
“Disarm”,Disarm
```

The first item in each entry is an action name and the second is the name of a function. Convention and common sense dictate that the action and function names should be the same, or at least related in a well-defined way.

You can change the translation table of any widget within Sun WorkShop Visual. You cannot change the action table of a widget. However, you can define new actions in an application-global action table which is searched after the one associated with the widget. This requires some coding, as described below.

Note – Translations are not supported on Microsoft Windows. For this reason, the Apply button in the Translations dialog turns pink when in Microsoft Windows mode.

Modifying a Translation Table

Modifying the translation table of a widget in Sun WorkShop Visual is straightforward. To understand the procedure, do the following simple exercise in Sun WorkShop Visual.

1. **Create a simple widget hierarchy containing a PushButton.**
2. **Select the PushButton icon in the widget hierarchy.**
3. **From the "Widget" menu, select "Translations...".**

   This displays the translations dialog, shown in Figure 6-7.

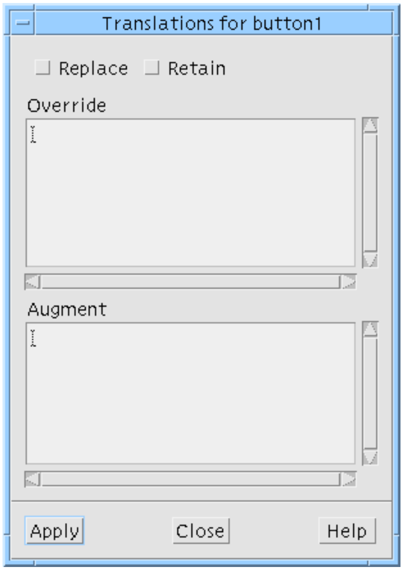


FIGURE 6-7 Translations Dialog

4. **Click in the lower section, under "Augment", and type**: **`Ctrl<Key>q: ArmAndActivate()`**
5. **Click on "Apply".**

This adds the new translation to the PushButton widget and you can now try its effect.

Note – You may notice an error dialog indicating that the action has not been found. This is for information only, the change has been taken. See "Additional Actions" on page 197 for information about creating an action table within Sun WorkShop Visual.

---

6. **Place the mouse pointer over the pushbutton in the dynamic display window.**
7. **Type: `<Ctrl-Q>`**

   The effect is identical to clicking with mouse button 1. Note that translations do not work if the window does not have the input focus and that the input focus behavior depends on the configuration of your window manager.

   You can also change the translations you have specified so that the button triggers on other events. Note that if you do this, the previous translation remains effective in addition to the new one until you reset the widget.

Augment, Override, and Replace

The translations dialog has sections labeled "Override" and "Augment". You can enter new translations in either section or both. They only differ if you specify a translation with the same event specification as an existing translation. If you type the new translation into the "Override" box, your new translation replaces the existing one. If you use "Augment", the existing translation takes precedence.

The existing default translations for the widget are not affected when you add translations unless you override them. This is important because Motif widgets have many default translations that produce their expected behavior.

If you set the "Replace" toggle, however, all existing translations are removed and replaced by the translations you enter. Use "Replace" with caution. Do not confuse "Replace", which removes all the default translations, with "Override", which replaces them one by one.

Translation Table Syntax

The syntax of translation tables is complex. The following sections detail the syntax as used in Sun WorkShop Visual. For a complete and definitive description, consult the X toolkit documentation.

Each entry in a translation table has the form:

```
[modifier_list]<event>[,<event>...][(count)][detail]:
[action([arguments])...]
```

Square brackets ([]) indicate that an item is optional; an ellipsis (...) indicates that the item may be repeated.

Modifier List

The *modifier list* represents the state (pressed or not pressed) of the modifier keys (such as Control and Shift) and the mouse buttons (X believes that a mouse has five buttons). The most useful modifiers are Ctrl, Shift, Alt and Meta. These can be abbreviated as c, s, a and m.

If the modifier list is omitted, the state of the modifiers is unimportant:

`<Key>Q` matches *<Q>*, *<Ctrl-Q>*, *<Alt-Meta-Q>*, etc.

If a particular modifier is not mentioned in the list, its state is unimportant:

`Ctrl<Key>Q` matches *<Ctrl-Q>, <Ctrl-Meta-Q>, <Ctrl-Alt-Meta-Q>*,...

You can specify multiple modifiers in the modifier list:

`Ctrl Meta <Key>Q` matches *<Ctrl-Meta-Q>* but not *<Ctrl-Q>* or *<Meta-Q>*

To specify that a modifier must not be pressed, precede it with a tilde (~):

`Ctrl ~Meta<Key>Q` matches *<Ctrl-Q>* but not *<Ctrl-Meta-Q>*

To specify that the modifiers pressed must exactly match what you specify, start the modifier list with an exclamation mark (!):

`!Ctrl<Key>Q` matches *<Ctrl-Q>* but not *<Ctrl-Meta-Q>* or *<Ctrl-Q>* with a mouse button pressed.

The modifier “None” means that there must be no modifiers pressed at all.

`None<Key>Q` matches *<Q>* but not *<Ctrl-Q>* or *<Alt-Meta-Q>*, etc.

Normally, translations are not case-sensitive. `<Key>Q` matches both *<Q>* and *<q>*. You can specify that a translation is case-sensitive by preceding it with a colon (:).

`:<Key>Q` matches *<Q>* but not *<q>*

Event and Count

The *event* can be the name of an X event, or one of a number of aliases. Some of the most useful events are *Key* (a key press), *BtnDown* and *BtnUp* (for any mouse button) and *Btn*N*Down* and *Btn*N*Up (*where N is between 1 and 5). For a complete list of events and aliases, see the Xt documentation.

`<Key>a` matches <*a*>

`<Btn1Up>` matches a release of mouse button 1

You can specify a sequence of events in a translation, separated by commas.

`<Key>Q,<Key>A` matches <*Q*> followed by <*A*>, with no intervening event.

`<Btn1Down>,<Btn1Up>` matches a click of mouse button 1.

The count can be used with button press and release events to detect multiple clicks. The count is a number from 1 to 9, possibly followed by a plus (+).

`<Btn1Down>(2)` matches two presses of mouse button 1

`<Btn1Up>(3+)` matches 3 or more releases of mouse button 1

If a count is used, the button events must come close together (usually within 200 milliseconds of each other), or there is no match.

Detail

The final field in the event specification is the *detail*. This is normally used only with key events, where the detail specifies which key is to be pressed.

The value specified in the detail field is a *keysym*, as in the header *<X11/keysymdef.h>*, with the *XK_* prefix removed. For most keys, this is the same as the character on the key.

`<Key>a` matches <*a*>

For non-alphanumeric keys, check the name of the keysym. The keysym for "+" is *XK_plus*, so

`<Key>plus` matches <+>

Since matching is case-insensitive, this also matches the other symbol on the plus key, which is <=> on most keyboards.

Motif adds another level of complexity by translating certain incoming key events into Motif virtual keysyms. You should use these virtual keysyms in your translation tables instead of the X ones.

`<Key>osfDelete`, not `<Key>Delete`

The virtual keysyms are listed below. For details of their interpretation, see the *VirtualBindings(3X)* section of the *Motif Programmer's Reference*.

TABLE 6-1 OSF Virtual Keysyms

osfActivate	*osfAddMode*	*osfBackSpace*	*osfBeginLine*
osfCancel	*osfClear*	*osfCopy*	*osfCut*
osfDelete	*osfDown*	*osfEndLine*	*osfHelp*
osfInsert	*osfLeft*	*osfMenu*	*osfMenuBar*
osfPageDown	*osfPageLeft*	*osfPageRight*	*osfPageUp*
osfPaste	*osfPrimaryPaste*	*osfQuickPaste*	*osfRight*
osfSelect	*osfUndo*	*osfUp*	

You can also use the detail field with mouse button events to specify a particular mouse button. This is not commonly done since it is easier to specify the mouse button in the event field.

`<BtnDown>Button1` is the same as `<Btn1Down>`

Actions

The actions on the right side of the translation table entry are simple. Usually each action is just a name followed by parentheses. Although any number of string arguments can be given between the parentheses, most action routines expect no arguments. Arguments should not be quoted. Typical additional translations for a ScrollBar widget might be:

```
<Key>d:IncrementDownOrRight(0)
<Key>u:IncrementUpOrLeft(0)
```

You can specify multiple actions or none at all. Overriding an existing translation with one that has the same event specification but no action is a useful way of disabling part of a widget's default behavior.

In many cases, the actions used are the ones predefined by the toolkit. The "Additional Actions" on page 197 discusses how to add your own actions.

Translation Table Ordering

When an event is received, the translation table is searched from the top down. The search terminates at the first entry whose event specification matches the event. This means you should organize your translation table with the most specific events first. For example, a translation table might contain the following entries:

```
<Key>q: action1()
Ctrl<Key>q: action2()
```

When the user types either *<Q>* or *<Ctrl-Q>*, the search terminates at the first entry and *action1()* is invoked in both cases. To make *<Ctrl-Q>* invoke action2, you must reverse the order of the entries.

For additional subtleties in ordering translation tables, see the X toolkit documentation.

Available Actions

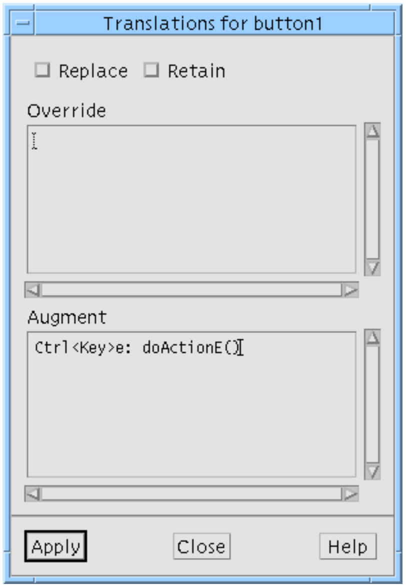
By changing the translation table, you can make a widget perform actions in response to event sequences that would not normally trigger those actions. While you can write your own action routines, translations provide the most benefit when you can use one of the built-in actions of the widget.

The built-in actions of the Motif toolkit are documented in the *Motif Programmer's Reference*. Each widget description includes both the default translations and the actions they invoke. Some of the primitive widgets offer a particularly large set of actions.

If you add a translation that uses one of these actions, you can test it in Sun WorkShop Visual immediately. Alternatively, a few built-in actions, such as the PushButton's *Activate()* action, invoke the functions in one of the widget's callback lists. In this case, it may be easier to specify a translation to call that action on the appropriate event sequence and put the code in an ordinary callback function.

Additional Actions

If you cannot find a built-in action to suit your needs, you can write your own *action routine* to perform the action. Figure 6-8 shows an example translation. The example shown says that when <*Ctrl-e*> is pressed, the action “doActionE” is triggered.



```
Ctrl<Key>e: doActionE()
```

FIGURE 6-8 Translation Example

If you define your own action routine, it needs to be added to the application’s “action table”. Sun WorkShop Visual does this for you automatically when you define the action procedures in the Action Procedures dialog, displayed by selecting “Action Procedures...” from the Module menu, as shown in Figure 6-9.

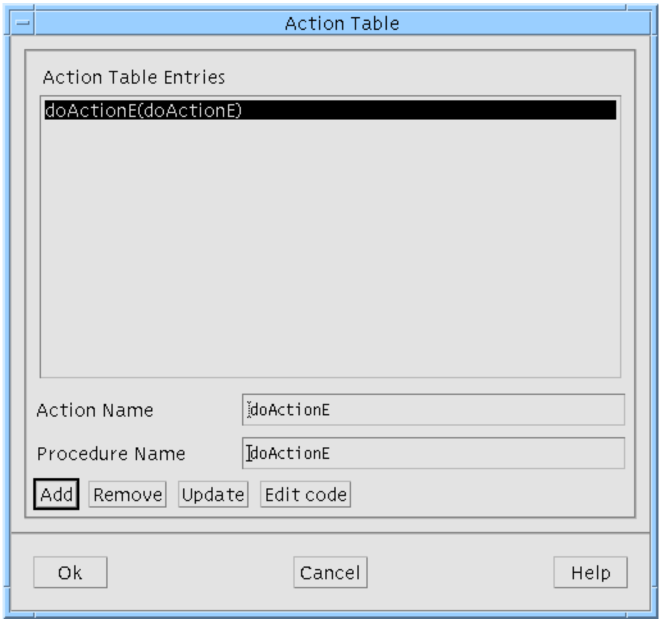


FIGURE 6-9 Action Procedures Dialog

In the example shown in Figure 6-8, “doActionE” is an action which needs to be defined for the application. Figure 6-9 shows the definition of this action. The “Action Name” is the name used in the Translations dialog, the “Procedure Name” is, as you might expect, the name of the procedure which will be called. In the interests of clarity, these names are usually the same.

Action procedures do not have any client data associated with them, but they are passed the parameters defined for the action in the Translations dialog. You can provide any number of parameters in that dialog and within the parentheses, so long as they are all strings.

You may have any number of action procedures defined within your design. Sun WorkShop Visual generates their stubs and the associated action table into the main code file.

Xt Procedures

Sun WorkShop Visual allows you to add the following types of Xt procedure:

1. **Additional event procedures**. This includes input sources and timeouts which may be treated as though they are the source of events, as well as Xt Work procedures which are called when there are no events to process.

2. **Language procedures**. One of these is called at the beginning of an X application as a means of customizing the localization of the application, although you may specify any number of them.

3. **Event handlers**. These procedures are called when one of a number of pre-defined actions occurs (e.g. “mouse button 1 pressed”). These bypass the translation table.

The first two types of Xt procedure listed above are specified on an application-wide basis. The fourth is specified on a per-widget basis. All are described in the following sub-sections.

Alternate Event Sources for X

An Xt (X toolkit) application normally waits for events from the X server. User actions, such as keyboard presses and mouse clicks, arrive at the Xt application via the X server. If, for some reason, a user is sitting quietly and not typing or clicking, then the application just waits. This means that an Xt application can spend a considerable amount of time waiting. For this reason, Xt allows you to register procedures to be called when there are no other events to process. You may also register procedures which respond to certain events not originating from the X server. Here are the “extra” procedure types which may be added:

1. Xt Work Procedure.

2. Input Procedure.

3. Timeout Procedure.

As with widget callback procedures added inside Sun WorkShop Visual, any Xt procedures that you have added are generated as stubs into the stubs file. You may then edit them from within Sun WorkShop Visual in the same way as widget callbacks may be edited. The code for adding your procedures is generated by Sun WorkShop Visual into the main module.

Note – If you are not generating a main module, your Xt procedures will not be added to your application.

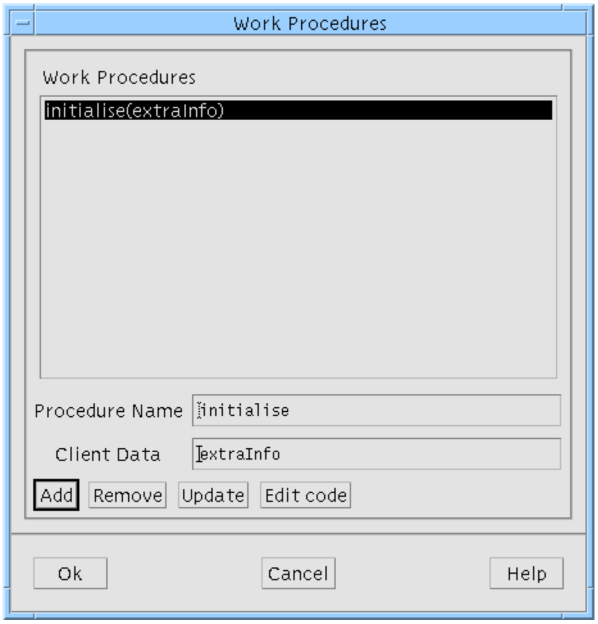
These procedures are added by selecting the corresponding item in the Module menu. The following sub-sections describe each procedure type individually.

Xt Work Procedures

A work procedure is a function that is called when Xt has no other events to process. Work procedures are, therefore, a convenient means of setting up a background batch process without interfering with events from the X server. One common use of work procedures is in program initialization. There are often many widgets which need to be created when an application is started. Since this is a time-consuming process, using work procedures allows the application to respond to user actions almost immediately.

Work procedures should return True or False. A value of True indicates to the X toolkit that your procedure should be removed from the work queue once executed. False indicates that it should be called again the next time the queue is empty of user events. You can have any number of work procedures registered at any time.

To add a work procedure, select “Work Procedures...” from the Module menu. The dialog shown in Figure 6-10 appears. You may add any number of work procedures but you should remember that your application cannot handle any other event while executing a work procedure. For this reason, work procedures should return quickly. Priority is generally given to work procedures in the order registered.



initialise(extraInfo)

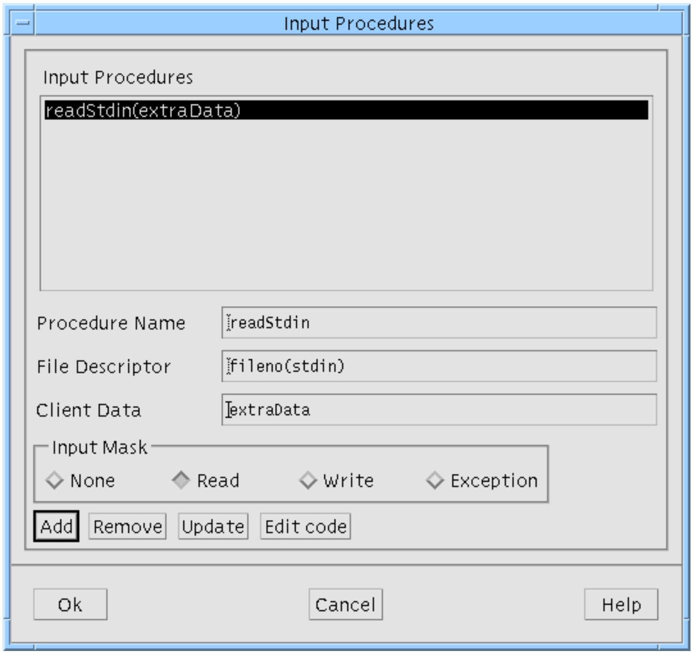
Procedure Name: initialise

Client Data: extraInfo

FIGURE 6-10 Work Procedures Dialog

Input Procedures

Use input procedures to set up a file or pipe as a source of events. The input procedure is called when the file is ready for reading (or writing). To add an input procedure, select “Input Procedures...” from the Module menu. The dialog shown in Figure 6-11 is displayed.



Input Procedures

readStdin(extraData)

Procedure Name: readStdin

File Descriptor: fileno(stdin)

Client Data: extraData

Input Mask: ◇ None ◆ Read ◇ Write ◇ Exception

FIGURE 6-11 Input Procedures Dialog

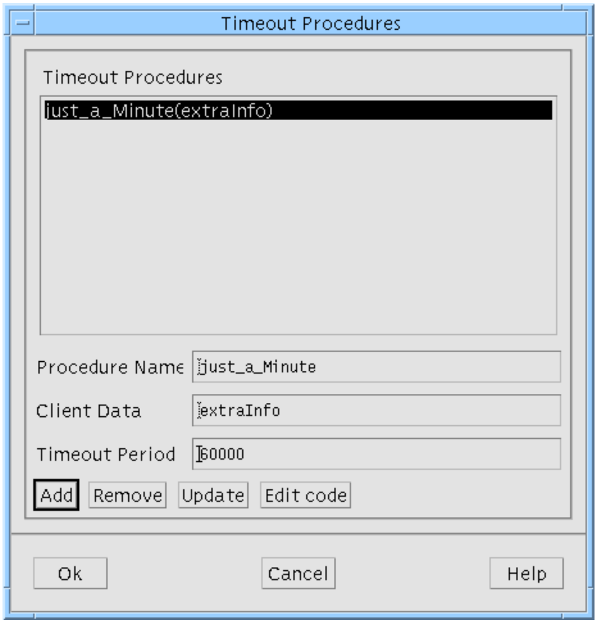
To define an input procedure, you will need to provide the file descriptor of the file or pipe. The Input Mask specifies the type of access the file should have. When the file is ready for the specified access, the input procedure is called. You may define any number of input procedures, they will be added in turn.

Timeout Procedures

A timeout procedure provides a means of performing a function after a specified amount of time has elapsed. Timeouts are called once only, so if you need a timeout procedure to be called at regular intervals, it will have to add itself as another timeout procedure before exiting. The following function call will do this:

```
XtAppAddTimeOut(appContext, timeoutPeriod, procedure, clientData);
```

To add a timeout procedure, select “Timeout Procedures...” from the Module menu. This causes the dialog shown in Figure 6-12 to be displayed.



Timeout Procedures

just_a_Minute(extraInfo)

Procedure Name: just_a_Minute

Client Data: extraInfo

Timeout Period: 60000

FIGURE 6-12 Timeout Procedures Dialog

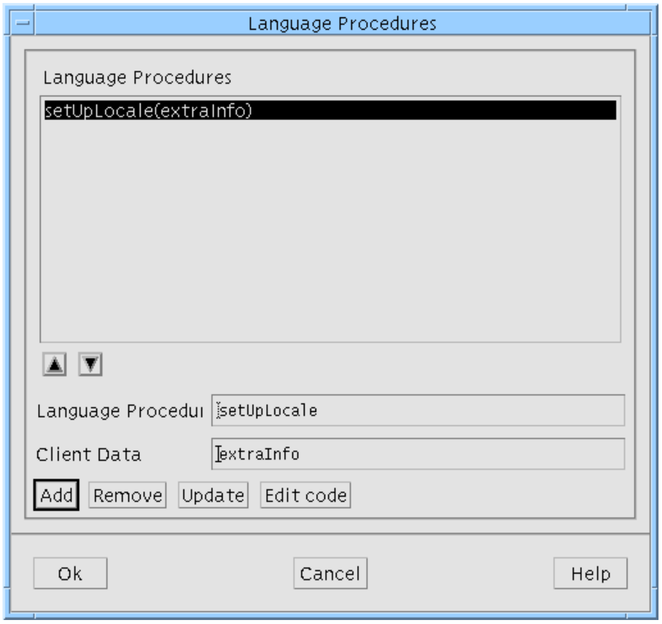
The “Timeout Period” is specified in milliseconds.

Timeout procedures work better in an X environment than time-interrupt programming using signals, which is often the preferred method with UNIX. You may define any number of timeout procedures; Sun WorkShop Visual generates them all into the main code file.

Language Procedures

An application operates within the context of a particular locale. The locale determines how to accept keyboard input, how to display characters and the format of date and time strings. This allows developers to customize their applications for use in different countries.

Sun WorkShop Visual generates a call to the X toolkit routine `XtSetLanguageProc` in the main code file. One of the parameters to this routine is the name of the procedure which will set up the locale. Xt provides a default language procedure, but you can define your own should you wish to have additional methods of setting the locale or only provide support for certain locales. To define a language procedure, choose “Language Procedures...” from the Module menu. The Language Procedures dialog, shown in Figure 6-13, is then displayed.


Language Procedures

Language Procedures

setUpLocale(extraInfo)

Language Procedu setUpLocale

Client Data extraInfo

FIGURE 6-13 Language Procedures Dialog

You may specify as many language procedures as you like, but only one will be effective (as only one can be passed as a parameter to `XtSetLanguageProc`). Sun WorkShop Visual chooses the procedure at the top of the list in the Language Procedures dialog as the procedure to pass in. You can change which is the topmost procedure in this list by using the arrow buttons underneath. Simply select the required procedure and press the up arrow until it is at the top. Stubs for all language procedures in the list are generated into the stubs file.

Event Handlers

Event handlers provide an efficient means of performing low level input handling which bypasses the translation tables of the widget. They are particularly suited to high-volume events. A translation and associated action, however, can do almost anything that an event handler can do but may be easier to maintain.

Event handlers, unlike the other Xt procedures, are defined for individual widgets rather than for the whole application.

Add an event handler by selecting the widget whose events are wanted and then choosing “Event Handlers...” from the Widget menu. The dialog shown in Figure 6-14 is displayed.

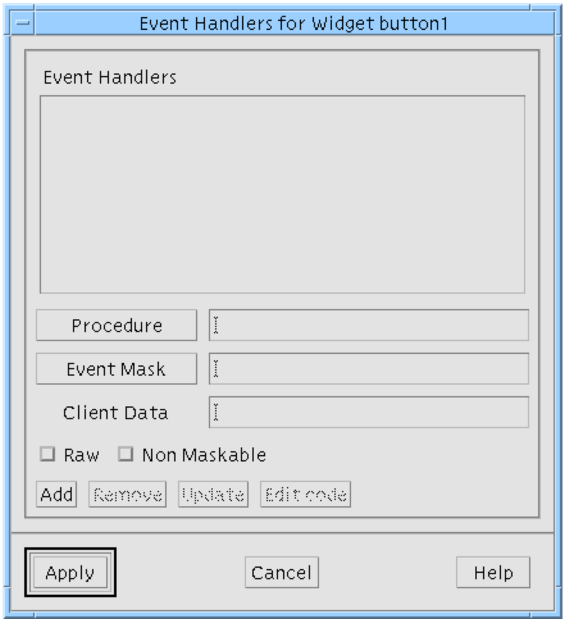


FIGURE 6-14 Event Handlers Dialog

In this dialog there are text fields for entering a procedure and an event mask. The text fields have corresponding buttons which, when pressed, display sub-dialogs showing the procedures and event masks available. You may define a new procedure but the event masks should come from the valid set displayed. The event masks are annotated with the Java “coffee cup” if they are applicable to Java code, and indicated with a tick if they are mapped to MFC for Windows, as shown in Figure 6-15.

Note – You will only see the ticks and crosses for MFC mapping if you are running in Windows mode. More information on Event Handlers for MFC is given in

---

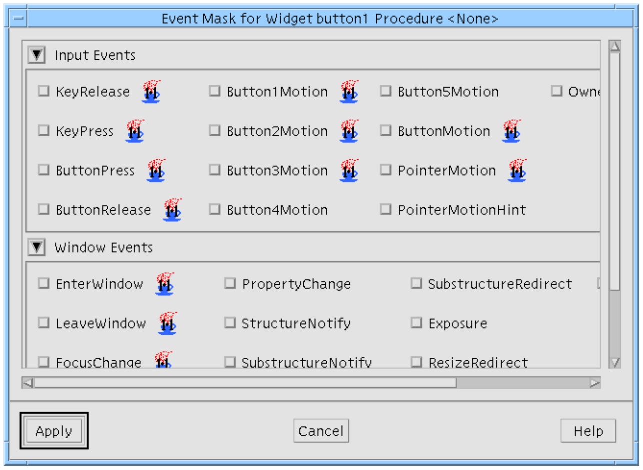


FIGURE 6-15 Event Masks

If the “Non Maskable” toggle is set, the event handler will also receive the non-maskable events (ClientMessage, GraphicsExpose, MappingNotify, NoExpose, SelectionClear, SelectionNotify and SelectionRequest). This toggle would normally remain unset as these events are not particularly useful.

The “Raw” toggle is a means of telling Sun WorkShop Visual to add a raw event handler. This is an event handler which does not respond immediately to the events for which it is registered. Instead, the handler is triggered when its events are selected elsewhere (by another event handler, for example).

One possible use of raw event handlers is to “shadow” another event handler. If both are added with the same event mask, but one is “raw” and the other is not, then both handlers will be called when the appropriate events occur. The raw event handler can then log the events that the other handler receives.

As with callback functions, you can specify the same event handler for any number of widgets and then use the Client Data to set the context.

Typically, event handlers are used in situations where the widget has little in the way of built-in interaction support, or where large volume or crude input may need processing. DrawingAreas are obvious candidates for event handlers.

Event handlers are generated by Sun WorkShop Visual into the code file as part of the general widget configuration and a stub is generated into the stubs file.

There is a “Widget annotation” for event handlers. By selecting the event handler annotation from the pullright menu in the View menu, you can see at a glance which widgets have had event handlers defined for them.

Editing Xt Procedures

As with callbacks, the generated stubs for Xt procedures can be edited from within Sun WorkShop Visual using your favorite configured editor.

CHAPTER 7

Generating Code

Introduction

Up to this point, you have used the interactive features of Sun WorkShop Visual to build a working prototype of a user interface. Now you can use the code generation features to produce the files necessary to convert that design into a free-standing program. Code can be generated as C, C++, Java or UIL.

In this chapter, you will:

- Generate a primary module for your design, including all the code needed to prototype your interface
- Generate a stubs file for convenience in writing callback functions
- Compile, link and run your prototype
- Generate an X resource file containing resource settings which are not hard-wired into the code
- Write your *quit()* callback

This chapter also includes an analysis of the code which is generated into the various files and a discussion of strategies for arranging your files.

Prerequisites

You need some knowledge of C, C++, Java or UIL to understand the generated code files and to supply code for callback functions. You also need some knowledge of the X Window System.

The Generate Menu

The Generate Menu is used to generate source code, X resource files and Makefiles from your design. The Generate Menu has seven items: “C”, “C++”, “UIL”, “X Resources”, “Makefile”, “Java” and “Generate”. The first six options generate the type of file selected provided that you have set up that file in the Generate dialog.

The “Generate” option displays the Generate dialog.

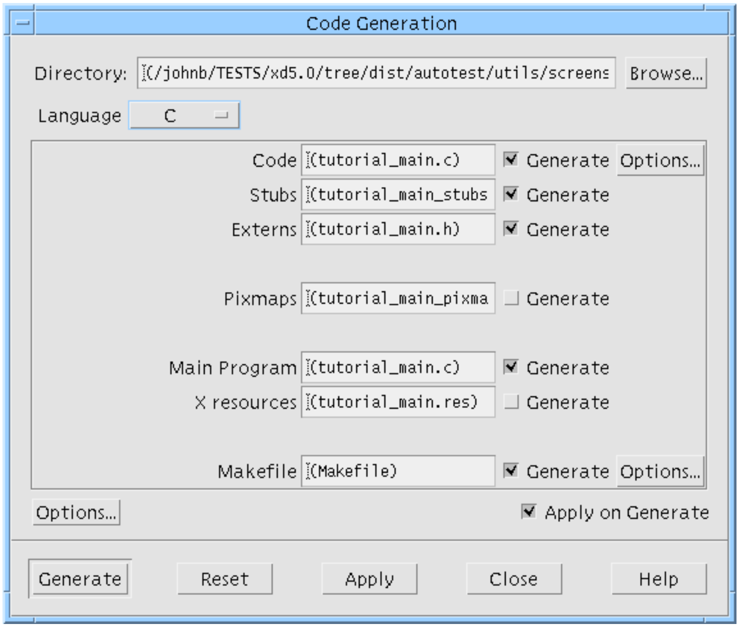
Note – In Microsoft Windows mode the Generate Menu has an additional selection to generate Microsoft Windows resource files. This is discussed in “Building the Application” on page 402.

C is used for the examples in this chapter. The procedure for generating C++ is exactly the same and you may use C++ for the tutorial if you prefer. The procedure for UIL is very similar to the procedure for C with the exception of one additional step which is discussed in “Special Notes for UIL” on page 210.

For information on generating Java, see Chapter 10, “Designing for Java”, starting on page 313.

Generate Dialog

To display the Generate dialog, pull down the Generate menu and select “Generate”.



Directory: (/johnb/TESTS/xd5.0/tree/dist/autotest/utils/screens

Language C

Code (tutorial_main.c) ☑ Generate

Stubs (tutorial_main_stubs ☑ Generate

Externs (tutorial_main.h) ☑ Generate

Pixmaps (tutorial_main_pixma □ Generate

Main Program (tutorial_main.c) ☑ Generate

X resources (tutorial_main.res) □ Generate

Makefile (Makefile) ☑ Generate

☑ Apply on Generate

FIGURE 7-1 Generate dialog

This dialog gives you an overview of all the files which can be generated from your design. The dialog contains some default settings, including default filenames. The default filename is appropriate to the type of file and the language being generated. You can change the defaults in the application resource file. See “Filters” on page 871 for more details. If, having generated code, you wish to reset the dialog back to the default filenames (complete with brackets) simply press the “Reset” button.

Setting up the Dialog

Before setting up individual files to be generated, you need to set two options which affect all the files:

- The language you are using.
- The base directory in which you wish to generate your files.

Setting the Language

For the language, use the “Language” option menu at the top of the dialog. You have a choice of C, C++, UIL and Java. When in Microsoft Windows mode, you have the additional choices “C++ (Motif XP)” and “C++ (Microsoft Windows MFC)”. See

Chapter 11, “Designing for Microsoft Windows”, starting on page 357 for more details about Microsoft Windows mode, Motif XP and MFC. This tutorial uses the C language, so:

- **Make sure that “C” is selected from the “Language” option menu.**

Note – Use of the Generate Dialog is quite different when “Java” is the selected language. For this reason, the Generate Dialog for Java code generation is explained in “Generate Dialog” on page 336 in Chapter 10 ”Designing for Java“.

Special Notes for UIL

When you work in UIL, the code generation procedure is basically the same as for C. However, because UIL is not as powerful a language as C, there are some features of Sun WorkShop Visual which cannot be implemented in UIL. To get the full functionality of your design, a supplementary C file must be generated in addition to your UIL file.

When you select “UIL” from the Language option menu, enter the name of the UIL file in the “UIL” field and the name of the supplementary C file in the “Code” field. You must also specify a name for the compiled UIL file. This is done by pressing the Code “Options” button and then entering the name in the “Uid file” field.

“Includes” (see “Setting up the Primary Source File” on page 211), “Main program”, and “Links” (see “Code Generation Options” on page 218) can be generated into the C file but not into the UIL.

UIL is a Motif-specific language and does not work with widgets outside the Motif toolkit. If your design contains a widget from another toolkit, you must use C or C++.

Setting the Base Directory

To set the base directory you can type it directly into the text box labelled “Directory” or you can press the “Browse” button. The “Browse” button displays a file selection box so that you can find and select a directory. The filenames of the files to be generated are *relative* to the base directory. By default, the base directory is the directory from which you last opened a saved design or, if you are working on a new design, the directory from which you invoked Sun WorkShop Visual. The default, which is shown enclosed in brackets, is not saved into your “.xd” file. An explicitly named directory will be saved.

You can type an absolute pathname into the filename text boxes. This method is not recommended because you would have to do this for every file. It is easier and less prone to typing mistakes if you set up the directory first and assume all files are relative to that directory.

Setting up the Primary Source File

In the text box labelled “Code”, enter the filename of your primary source file. A description and explanation of this file is given in “Analysis of the Primary Module” on page 230.

1. **Type: `icecream.c` into the text box labelled “Code”.**

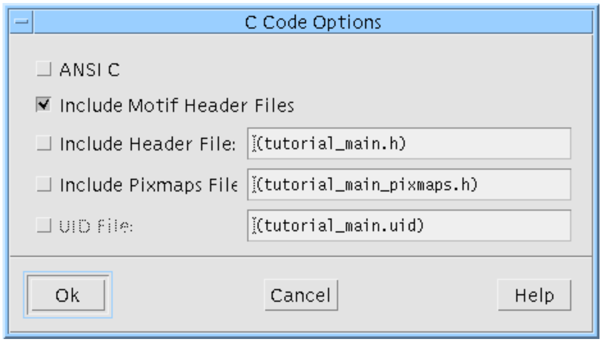
   By convention, all C files, including primary source files and stubs files, have the suffix `.c`.

2. **Check that the “Generate” toggle next to the text box is set.**

   Only those files which have the “Generate” toggle set will be generated.

3. **Press the button labelled “Options” beside the “Code” filename.**

   This button displays the options relevant to the primary source file.



C Code Options

□ ANSI C

☑ Include Motif Header Files

□ Include Header File: (tutorial_main.h)

□ Include Pixmaps File (tutorial_main_pixmaps.h)

□ UID File: (tutorial_main.uid)

FIGURE 7-2 Primary Source File Options Dialog

The Options Dialog for the primary source file offers the following:

- **ANSI C** Select this if you wish Sun WorkShop Visual to generate ANSI C.
- **Include Motif Header Files**. This is selected by default. If you deselect this option you must find somewhere else to include the Motif headers, otherwise you will see compilation errors. For the tutorial, check that this option is selected. This option is not visible if you are running in Microsoft Windows mode and you are generating MFC flavor code.

- **Include MFC Header Files**. This option is only visible when you are running in Microsoft Windows mode and you are generating MFC flavor code. This is selected by default and ensures that your code file includes the header files required if you wish to use the MFC.
- **Include Header File**. If you wish Sun WorkShop Visual to add a line to your source file to include a header file[1], you would set the toggle next to this text and type the name of the header file in the text box. For the tutorial, however, we are not going to include a header file so make sure that this option is *not* selected. See "Notes on Including a Header File" on page 213 for additional information on including header files in your source code.

Note – If you generate a separate code and main program file, you must include the generated externs file so that both files can use the same variables.

- **Include Pixmaps file**. This is similar to the "Include Header File" option. If you are going to generate a separate pixmaps file, and you wish to include the generated pixmaps file in your primary source file, turn on this toggle and supply the name of the pixmaps file in the text box. Sun WorkShop Visual then generates an *#include* directive instead of explicit pixmap definitions in the primary source file. See "Setting up the Pixmaps File" on page 214 for details on generating a pixmaps file. For the tutorial, make sure that this option is not selected.
- **Uid File**. This item only appears if you are generating UIL. This is the name of the Uid file. See "Special Notes for UIL" on page 210 for more information.

4. **Press "Ok" to close the Options dialog.**
5. **Press "Apply" in the Generate dialog.**

   This will save everything you have set up without generating straight away.

#include Generation Control

As part of your design, you can specify the names of files to be added as include files in the generated code. By default, Sun WorkShop Visual generates these include statements with angled brackets, as in the following example:

```
#include <externs.h>
```

This behavior is not suitable for all language types and for all configurations. The following resources now allow you to control the way in which "#include" statements are generated:

```
visu.defaultIncludeInQuotes: true
visu.defaultIncludeObjectFileInQuotes: false
```

[1] The terms "Header file" and "Externs file" are used interchangeably in both Sun WorkShop Visual and the User's Guide.

The first statement controls whether the external files are included in quotations or not, the second controls the generation of header files which are automatically generated by Sun WorkShop Visual (the `xdclass.h` files).

This mechanism overrides the *default* behavior. Explicitly placing "" or <> around the name of the header file when you type it into the Primary Source File Options dialog overrides these defaults.

Notes on Including a Header File

If you wish to include both an external header file (not created by Sun WorkShop Visual) and an Externs file created by Sun WorkShop Visual in your code file, you can enter the name of the external header file in the “Include Header File” field and then make sure that this header file includes the Externs file generated by Sun WorkShop Visual.

When you set the “Include Header File” toggle in the Code Options dialog and enter the name of the externs file, the file is included in the code file, the stubs file (if you are generating one) and the main program file (if it is different from the code file).

If you are generating a separate code and main program file, you *must* include the externs file because they both use variables which are defined in the externs file.

Setting up the Stubs File

Callback stubs, i.e. “empty” routines with the specified callback or method name, are generated for all callbacks and callback methods in your design. These are generated into a separate source file called a *Stubs File*. If you have callbacks or methods in your design, generating a stubs file allows you to compile the application since the callbacks are referenced from the main source code. It is left to you, however, to add the required functionality to the callback routines. You are shown how to do this in “Adding Callback Functionality” on page 226.

Along with the callback stubs, Sun WorkShop Visual generates special comments. These comments are explained in “Stubs File Comments” on page 229.

Your design has just one callback: *quit()*. For now, generate a stubs file with an empty *quit()* function. The dummy function lets you compile, link and run your application as a prototype. Later (in “Adding Callback Functionality” on page 226), you will add functionality to *quit()* to complete your application.

1. **Type: `stubs.c` in the text box labelled “Stubs”**.
2. **Set the “Generate” toggle next to the text box.**

   Remember that only those files with a selected “Generate” toggle will be generated.

   There are no separate options for the Stubs file.

Setting up the Externs File

Sun WorkShop Visual can generate a header file with *extern* declarations for all widgets which are global in scope, C++ class definitions and C structure definitions for your design. Global widgets include all widgets which you have explicitly named and those which you have designated as global on the Core resource panel.

To set up an Externs file, type the name of the file in the text box labelled “Externs” and set the “Generate” toggle next to it. By convention, Externs files have the suffix *.h*.

- **Make sure that the “Generate” toggle next to the text box labelled “Externs” is not set because you do not need an externs file for the tutorial.**

To include the generated Externs file in your primary module, see the description of the Options dialog in “Setting up the Primary Source File” on page 211. Sun WorkShop Visual then generates a *#include* directive instead of explicit type definitions in the primary source file. Global widgets are still allocated in the main source file when you do this.

The Externs file is also useful for including in your stubs file or other code files where you access global widgets or refer to type definitions. See “Global Widget Variables” on page 180 for more details.

There are no separate options for the Externs file.

Setting up the Pixmaps File

The Pixmaps file is similar to the Externs file. It is a header file with *static* declarations of all the pixmaps in your design. Generating one of these files lets you keep the cumbersome definitions of pixmap structures separate from your primary source file.

To generate a Pixmaps file, type the name of the file into the text box labelled “Pixmaps” and set the corresponding “Generate” toggle. By convention, Pixmaps files have the suffix *.h*.

- **Make sure that the “Generate” toggle next to the text box labelled “Pixmaps” is not set because you do not need a pixmaps file for the tutorial.**

Setting up the Main Program File

The main program file is a file containing the *main()* procedure. By generating a separate file you can keep this procedure away from the rest of your source code. This is useful if you need to edit the procedure to perform some of your own initializations or call other parts of your application before starting off the user interface. If you edit this file, make sure that you generate it only once as subsequent generations will overwrite your changes. A full description of the *main()* procedure is provided in “Description of the Main Program” on page 234.

If you wish the *main()* procedure to be generated into the primary source file, make sure that the text box labelled “Main program” contains the same name as the “Code” text box and the corresponding “Generate” toggle is set.

For the tutorial:

1. **Check that the base directory is set.**

   See “Setting the Base Directory” on page 210.

2. **Type: `icecream.c` into the text box labelled “Main Program”**.

3. **Check that the “Generate” toggle next to the “Main Program” text box is set.**

   There are no separate options for this file.

Setting up the X Resource File

As you have seen in Chapter 3 ”Resources“, resource settings need to be available or they are not applied when your interface runs. You can make them available in one of two places: the primary *source file* or the *X resource file*. Generating resources into the source file is known as *hard-wiring* them.

1. **Check that the base directory is set. See “Setting the Base Directory” on page 210.**

2. **Type into the text box labelled “X resources” the filename:** `icecream.res`

3. **Set the “Generate” toggle next to the text box.**

   You can control which resources are generated and into which file by setting the options described in “Code Generation Options” on page 218.

Making X Find Your Resource File

X does not automatically recognize *icecream.res* as your application’s resource file. One recommended method of telling X where to find this file is to copy the resource file to the designated application resource directory (*/usr/lib/X11/app-defaults* on POSIX systems). The filename in that directory should be the application class name, *XDTutorial*, without a suffix (see “Code Generation Options” on page 218 for details on how to set the application class name). This method avoids any confusion of this application-specific resource file with other files you might be using.

Because you may not have access permission to the application resource directory, a different method is described here.

4. **Set the environment variable XENVIRONMENT to the filename of the resource file.**

   The exact syntax for doing this will differ depending on which shell you are using. For a C shell, enter:

   ```
   setenv XENVIRONMENT icecream.res
   ```

   For a Bourne shell, enter:

   ```
   XENVIRONMENT=icecream.res export XENVIRONMENT
   ```

   There are other ways to get X to recognize your X resource file. To find out what they are, you will need to look them up in a book about the X Window System. See Appendix E for the names of some books you may wish to try.

Setting up the Makefile

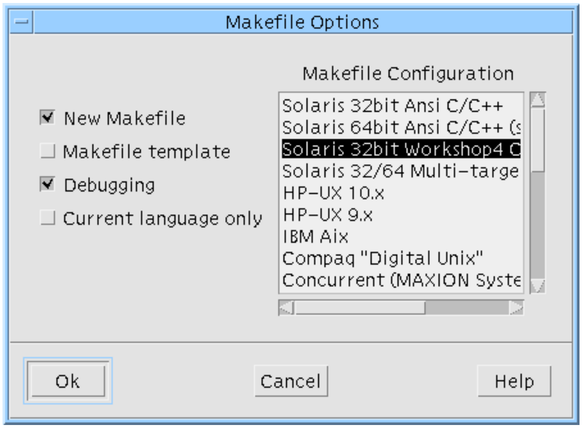
Sun WorkShop Visual can generate a makefile containing compilation instructions for all the files required by your design with the correct dependencies. For the tutorial, your makefile needs to compile both *icecream.c* and *stubs.c*. It also needs to link the resulting object files with the required libraries.

1. **Check that the base directory is set to the directory where your primary source file and stubs file have been or will be generated.**

2. **Type “Makefile” into the text box labelled “Makefile” and check that the “Generate” toggle is set.**

3. **Press the button labelled “Options” next to the Makefile text box.**

   This displays the Makefile Options dialog, shown in Figure 7-3.



☑ New Makefile
□ Makefile template
☑ Debugging
□ Current language only

FIGURE 7-3 Makefile Options Dialog

In order to generate a Makefile, Sun WorkShop Visual uses an internal makefile template which contains information about different platforms and environments. Using the template mechanism, Sun WorkShop Visual is also able to create a Makefile which can build the sources generated from more than one design into one application.

The “New Makefile” and “Makefile Template” toggles in the Makefile Options dialog relate to the two different types of makefile that you can generate: a simple makefile, which just builds the sources from one design, and a makefile with templates, which allows further sources to be added to it. “Debugging” refers to the “-g” flag for the compiler.

When you generate code in one language and then generate another set of code files in another language, all Makefiles generated afterwards will contain rules for both sets of generated files. Setting the “Current language only” toggle ensures that you are generating a Makefile for the current language only.

The right of this dialog shows the list of target platforms and compilers for which Sun WorkShop Visual can automatically generate a Makefile. This list also gives you the option of compiling a 64-bit application. All aspects of this dialog are explained more fully in Chapter 19, “Makefile Generation”, starting on page 553.

This section shows you how to generate a simple Makefile capable of building the sources from your tutorial design. For more details on Makefile generation, including using templates and customization of Makefiles, see Chapter 19, “Makefile Generation”, starting on page 553. For details on how to change the Makefile template application resource, see “Generation” on page 875 in Appendix D.

1. **For the tutorial, set the “New Makefile” toggle but do not set the “Makefile Template” toggle.**

   You are not going to add any more source files to the tutorial application so you do not need the template comments.

2. **You may turn off the “Debugging” toggle if you prefer.**

   We shall not be using debugging in this tutorial.

3. **Check that the selected item in the list on the right of the dialog matches your target platform and compiler.**

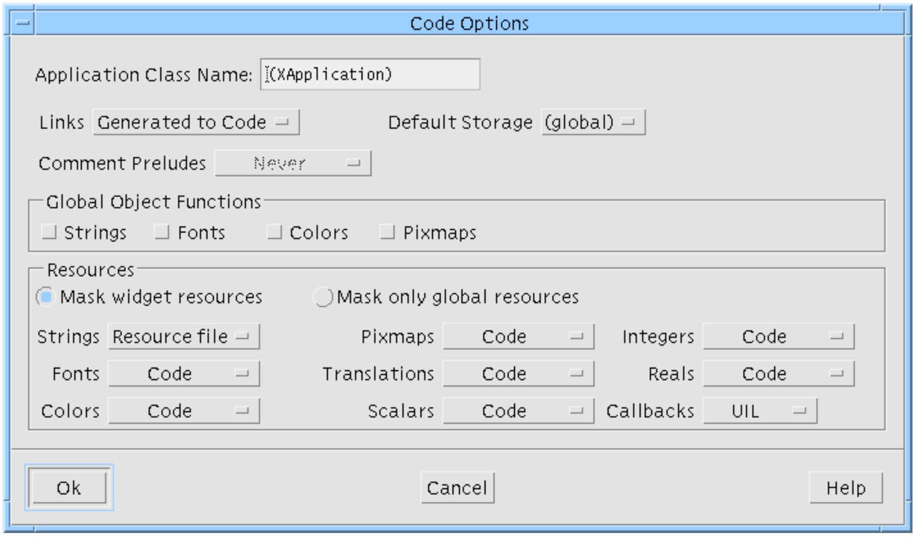
   The item selected by default should be appropriate. If you are not sure, ask your system administrator. More information on this list is given in “The List of Makefile Types” on page 555.

4. **Press “Ok” to save your changes and close the Makefile Options dialog.**

   This takes you back to the Generate dialog.

Code Generation Options

Before generating any files, you should check to see whether you need to change any of the code generation options. Press the “Options” button at the bottom of the Generate dialog. The dialog in Figure 7-4 appears.



Code Options

Application Class Name: (XApplication)

Links: Generated to Code

Default Storage: (global)

Comment Preludes: Never

Global Object Functions

□ Strings □ Fonts □ Colors □ Pixmaps

Resources

● Mask widget resources ○ Mask only global resources

Strings	Resource file	Pixmaps	Code	Integers	Code
Fonts	Code	Translations	Code	Reals	Code
Colors	Code	Scalars	Code	Callbacks	UIL

FIGURE 7-4 Generate Options Dialog

This dialog lets you control where, and whether, the code for links is generated, specify the application class name and control where individual resource types should be generated. These are explained separately below.

Application Class Name

The *application class name* is used to identify resource settings when you generate an X resource file. Assigning a name other than the default “XApplication” prevents confusion of your resource values with system-wide X resources.

Links Code Generation

Using the option menu labelled “Links”, you can choose whether the links are generated into the primary source file, into the stubs file or not generated at all. Alternatively, you could choose to generate the links code declarations only.

The code created as part of the links functionality consists of a set of generic functions (one for each type of link) and the declarations of these functions. Because the functions are always the same, you only need to generate one set of the functions for your whole application. If you are generating code from more than one design, each of which contains links, you only need to generate the functions once but you will need to generate the declarations for each design. The declarations are always generated into the primary source file.

Global Object Functions

Sun WorkShop Visual allows generation of global accessor functions for color, font, string and pixmap objects. This means that these objects can be shared across different files in your application. The toggles in the Code Options dialog allow you to make color, font, string and pixmap objects global. When selected, these toggles cause Sun WorkShop Visual to create one function per object which returns the object. Sun WorkShop Visual uses the name of the object to build a function name using the following algorithm:

```
xdGet + <object_name> + <object_type> + Object
```

where <object_name> is the name you bound to your object and <object_type> is one of “Color”, “Font”, “String”, “Pixmap”, depending on the type of the object. For example, a color object named “Foreground” would be generated by default with the accessor name:

```
xdGetForegroundColorObject
```

The xdGet tag can be overridden through the resource:

```
visu.globalObjectFunctionStart: xdGet
```

For example, suppose you have:

```
visu.globalObjectFunctionStart: my
```

and with a single color object “Background”, you would have the following code generated if the global object toggle for color is set:

```
Pixel myBackgroundColorObject() {
     ...
}
```

or, if the language is MFC:

```
CBrush myBackgroundColorObject() {
     ...
}
```

Control of Default Storage

By default, Sun WorkShop Visual generates each widget as a variable local to the function in which it is created. If you name a widget, however, Sun WorkShop Visual makes the variable global. This default behavior, for named widgets, can be controlled in the Code Options dialog by changing the “Default storage” option menu.

If a widget belongs to a class (i.e. is the child of a widget which has been made a class), however, it will remain a variable of that class, regardless of the storage option chosen in the Code Options dialog.

The default storage for named widgets is “global”. You can change this default by setting the following resource:

```
visu.defaultStorageType:static
```

The possible values of this resource are:

- default (that is, use the usual Sun WorkShop Visual default - in this case, “global”)
- global
- static

Comment Preludes

This option menu refers to the special comments which appear in the generated code around the areas where preludes can be added. Adding preludes is described in “Customizing the Generated Files: Preludes” on page 239. You can choose whether you wish comments to be added “Always” (whether a prelude has been specified or not), “When Defined” (i.e. only when a prelude has been specified) or “Never”. The default is “Never”.

The example below shows a comment for a shell pre-manage prelude. Because there is no prelude inside the comment we must assume that “Always” was the selected option:

```
/* visu: prelude for shell1: pre-manage >>> */


/* <<< pre-manage ends. */
XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
shell1 = XmCreateDialogShell ( parent, “shell1”, al, ac );
     ac = 0;
     /* visu: prelude for form1: pre-create >>> */


     /* <<< pre-create ends. */
     XtSetArg(al[ac], XmNautoUnmanage, FALSE); ac++;
     form1 = XmCreateForm ( shell1, “form1”, al, ac );
```

Control of Resources

The panel in the middle of the Options dialog contains references to each type of resource.

Motif defines the following resource types:

- Strings – Includes null-terminated character strings and *XmString*s
- Fonts
- Colors
- Pixmaps
- Translations – Defined by a command in the “Widget” menu and discussed in “Translations and Actions” on page 190
- Scalars – Multiple-choice resources, including Booleans. On most resource panels, these are on the “Settings” page
- Integers – Any integer resource setting which is not a scalar
- Reals – Any floating-point numeric resource value

You can specify into which file each resource type should be generated. If a resource is generated into the source code it is then *hard-coded* and cannot be modified through the resource file. Typically, any resources which are not generated into the source code are generated into the X resource file, where they can be edited by the end user. See “Setting up the X Resource File” on page 215 for more details on X resource files.

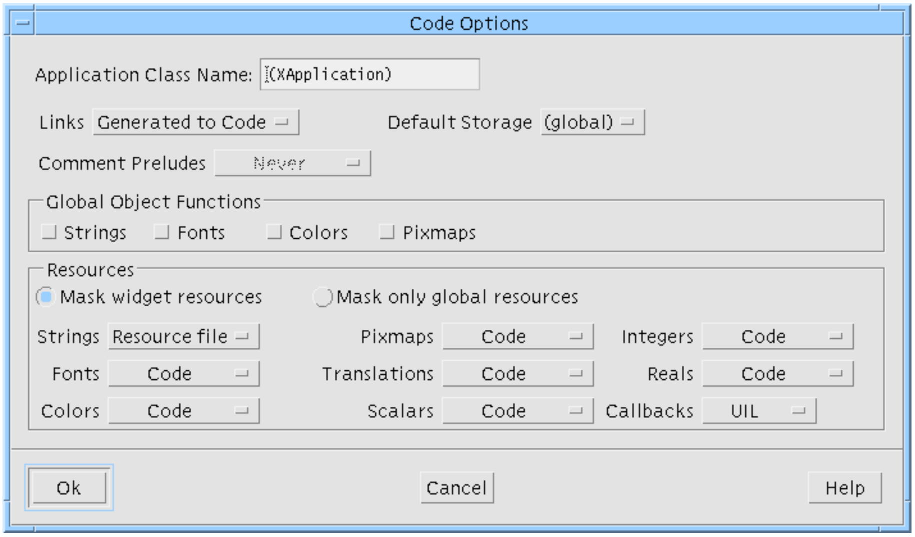
The option menu labelled “Callbacks” only appears if you have selected UIL as the language type. This lets you choose whether callbacks are registered in the UIL code or the C code. By default, they are registered in the UIL. If you use client data, however, you should generate the callbacks into the C code, because structure types cannot be defined in UIL. See “Special Notes for UIL” on page 210 for an explanation of the C for UIL file.

For the tutorial, make the following changes in the Generate Options Dialog:

1. **Select “Generated to code” from the “Links” option menu.**
2. **In the “Application Class Name” text box type: `XDTutorial`**
3. **Make sure that the resource type option menus are set as shown in Figure 7-5.**

   Note that the callbacks option menu is shown. This is only displayed if you have selected UIL as the language type.
4. **Make sure that the “Mask widget resources” radio button is on.**

   The significance of this radio button is discussed in “Masking Resources” on page 223.
5. **Click on “Ok”.**



Code Options

Application Class Name: (XApplication)

Links: Generated to Code

Default Storage: (global)

Comment Preludes: Never

Global Object Functions

□ Strings □ Fonts □ Colors □ Pixmaps

Resources

● Mask widget resources ○ Mask only global resources

Strings: Resource file

Fonts: Code

Colors: Code

Pixmaps: Code

Translations: Code

Scalars: Code

Integers: Code

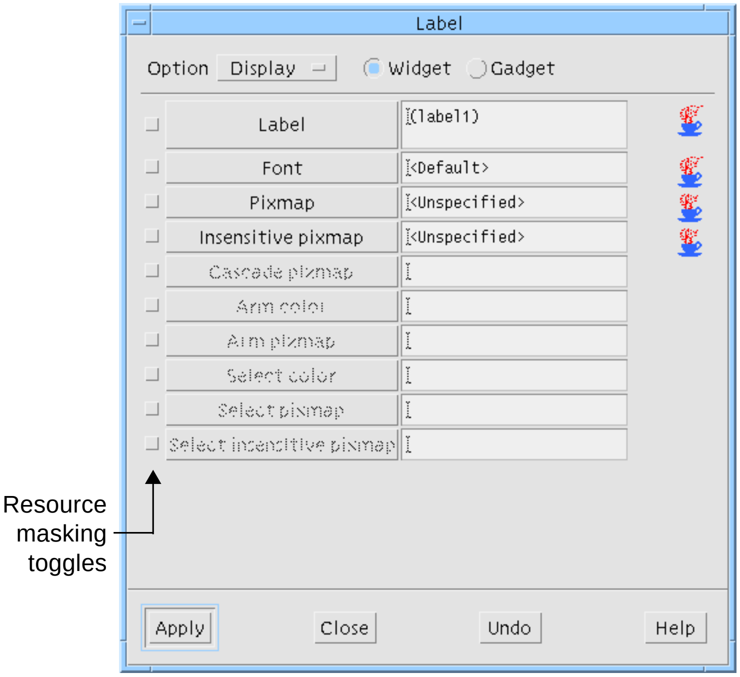
Reals: Code

Callbacks: UIL

FIGURE 7-5 Resource Settings for Code Generation (with UIL language type)

Masking Resources

If you look at any resource panel, you will see that it contains unlabeled toggles next to each resource, as shown in Figure 7-6.



□	Label	(label1)
□	Font	<Default>
□	Pixmap	<Unspecified>
□	Insensitive pixmap	<Unspecified>
□	Cascade pixmap	
□	Arm color	
□	Arm pixmap	
□	Select color	
□	Select pixmap	
□	Select insensitive pixmap	

FIGURE 7-6 Resource Panel Masking Toggles

These work in combination with the “Mask widget resources” and “Mask only global resources” radio buttons in the Generate Options dialog. Using these gives you control over the generation of resources on an individual basis.

Mask Widget Resources

The following description applies when the “Mask widget resources” toggle is set:

If an individual resource *does not* have its resource panel toggle set, the resource is generated according to the option menu for its type in the Generate dialog - i.e. a Label string with the resource panel toggle **off** will be generated to the file specified by the option menu labelled “Strings”.

If an individual resource *does* have its resource panel toggle set, the resource is generated to the *opposite* file from the one specified for its type by the option menu in the Generate dialog - i.e. an integer resource with the resource panel toggle **on** will be generated to the resource file if the “Integers” option menu is set to “Code” and to the code file if the option menu is set to “Resource file”.

Another way of saying this is that the option menus in the Generate dialog establish a general rule and the toggles in the Resource Panels identify exceptions to this rule.

Mask Only Global Resources

The following description applies when the “Mask only global resources” is set:

The option menus in the Generate dialog now apply only to global *objects* (font, color and pixmap objects). These (and only these) are controlled exactly as discussed for all resources in “Mask Widget Resources” on page 223.

All other resources are controlled *only* by their individual resource panel toggles. They are generated into the code file if the resource panel toggle is **off** and into the resource file if the resource panel toggle is **on**.

Examples of Masking Effects

In many cases, designers want to generate most strings into an X resource file so that they can be edited easily. This makes it possible to produce a foreign-language version of the application simply by editing the X resource file. To do this, you generate strings into the X resource file. However, there may be a few strings, such as the company’s address, which you do not want users to be able to change. You can hard-wire these few string resources by setting their individual masking toggles.

Similarly, you might want to let users edit nearly all color resources except for your company colors. To do this, set the masking toggles on the individual resources which control the company colors. When you generate code, generate colors as a group into the resource file. Sun WorkShop Visual hard-wires the tagged ones into the source code.

Default Settings

Default resource values, shown in brackets on the resource panels, are never generated into either file. In this case, Motif calculates the resource value at run time. The result may be different from the default value you saw while building the interface, depending on the platform you run the program on. Using default values is often helpful in making your application portable.

Finishing the Generate dialog

When you have finished setting up the files you wish to generate and their options in the Generate dialog, you can press “Apply” to save the settings or “Generate” to generate the files straight away. If you select “Apply” you can invoke the Generate dialog when you are ready and then generate all the selected files, or you can select the relevant button on the toolbar (or in the Generate menu) to generate individual files using the settings you applied earlier without producing the generate dialog again. If you have set the “Apply on Generate” toggle, pressing “Generate” performs a “Apply” as well as generating the requested files.

Running the tutorial

Having set up the Generate dialog with the files you wish to generate and their associated options, you are now ready to generate code and run your prototype application.

1. **To generate all the files for your prototype, display the Generate dialog and press the “Generate” button.**
2. **Make sure that you are in the directory where the files were generated, as specified in the Generate dialog.**
3. **Set VISUROOT to the path to the root of the Sun WorkShop Visual installation directory.**
4. **To build your prototype, type:** `make`
5. **Make sure that X can find your X resource file.**

   See “Making X Find Your Resource File” on page 215.
6. **To run your prototype, type:** `icecream`

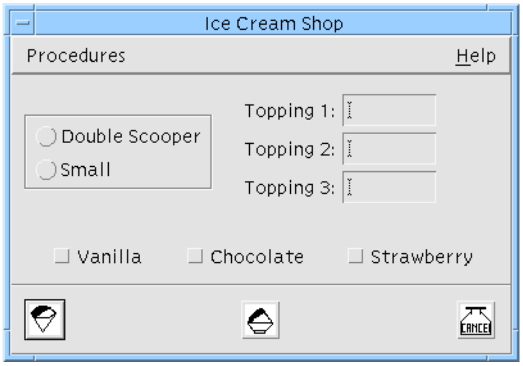


FIGURE 7-7 Interface Prototype Running

As in the dynamic display, all the widgets in your prototype are functional. You can click on the buttons, pull down the menus, and so on. Your prototype also includes links. You can display the help screen by pulling down the menu at the right side of the screen and clicking on its single entry; you can make the help screen disappear by clicking on its button.

Although your prototype also calls *quit()* when you click on *exit_button*, *quit()* doesn't do anything because you have not yet supplied the code to make it functional. This will be added in the following section. To terminate your prototype when you have finished examining it:

7. **Use the window manager to close the main window.**

Adding Callback Functionality

All that remains to complete the tutorial is adding functionality to *quit()*. Sun WorkShop Visual provides a means of editing callbacks directly in the stubs file.

Editing Callback Code from Within Sun WorkShop Visual

The Callbacks dialog contains a button labelled "Edit Code". Next to this button there is an option menu allowing you to tell Sun WorkShop Visual which language the stubs file is using. This is to prevent ambiguity if you are generating both C and

C++. You will not normally need to use this option menu as Sun WorkShop Visual will usually choose the correct value for you. See “Designating a Callback” on page 173 for a description of the Callbacks dialog.

Sun WorkShop Visual uses the Sun Workshop Edit Server when you press the “Edit code” button. The stubs file is opened in a separate editing window with the insertion point inside the function brackets of the selected callback. As you select other callbacks the insertion point moves around the file so that you are always at the correct place. If you try to regenerate the stubs file when there are unsaved changes in the Edit Server window, you will be prompted to save those changes first.

Before allowing you to edit the stubs file, Sun WorkShop Visual checks to see whether there have been any changes to the design since the last stubs file generation. If so, you are asked whether you wish to regenerate the file before editing. You cannot edit the file if you do not regenerate it.

If you have unsaved changes in an editing window when you try to regenerate the stubs file, you will be asked if you wish to save the file first. If you do save the file first, any callback code you have added will be preserved. See “Incremental Stubs File Generation” on page 228 for more details. If you choose to regenerate without saving changes, any changes made since the last save will be lost.

Editing the Callback

If you do not wish to use Sun WorkShop Visual to edit your callback:

- **Open `stubs.c` with a text editor and skip ahead to Step 5.**

To edit your callback from within Sun WorkShop Visual:

1. **Select the *exit_button* widget.**
2. **Click on the Callbacks button in the toolbar or select “Callbacks” from the Widget menu.**
3. **Select “quit()” from the list of callbacks.**
4. **Press the button labelled “Edit code”.**

   Pressing this button opens `stubs.c`.

Editing the Stubs File

The code for `quit()` looks like this:

```
void
quit (Widget w, XtPointer client_data, XtPointer xt_call_data )
```

```
{
    XmPushButtonCallbackStruct *call_data =
(XmPushButtonCallbackStruct *) xt_call_data;
}
```

5. **Replace the text between the braces of quit() by:** `exit (0);`

6. **Save the file.**

7. **Remake your executable and run the program.**

   Refer back to "Running the tutorial" on page 225 for details on how to do this.

   The "Exit" button is now functional. To quit your program:

8. **Pull down the Procedures menu of your interface and click on the "Exit" button.**

   "Callback Functions" on page 178 looks at callbacks in more detail, including the parameters passed to them and the ways in which you can access and manipulate widgets in your design.

   The tutorial is now complete. The remainder of this section looks in more detail at the issues discussed in previous sections.

Incremental Stubs File Generation

When you subsequently generate the same stubs file, Sun WorkShop Visual reads the special comments in that file in order to work out which callbacks and methods have already been generated. *In this way you can add your own code to the stub and it will not be overwritten*.

Sun WorkShop Visual then appends any new callbacks or methods to the end of the stubs file. Whenever a new stubs file is generated, the old version is copied to a file with the name you have specified and a `.bak` extension.

Note – When you remove a callback from a widget in Sun WorkShop Visual, the callback stub will remain in the stubs file. If you want to remove the routine, you will need to open the stubs file and remove the comment above the callback routine.

Below is the listing of a stubs file containing one callback, named quit:

```
/*
** Generated by Sun WorkShop Visual
*/
```

The Beginning of the Prelude:

```
/*
** Sun WorkShop Visual generated prelude.
** Do not edit lines before “End of Sun WorkShop Visual generated
prelude”
** Lines beginning ** Sun WorkShop Visual Stub indicate a stub
** which will not be output on re-generation
*/
/*
**LIBS: -lXm -lXt -lX11
*/
```

The End of the Prelude:

```
/* End of Sun WorkShop Visual generated prelude */
```

Special Comment to Indicate a Stub:

```
/*
** Sun WorkShop Visual Stub quit
*/
void
quit (Widget w, XtPointer client_data, XtPointer xt_call_data )
{
        XmPushButtonCallbackStruct *call_data =
                 (XmPushButtonCallbackStruct *) xt_call_data;
}
```

Stubs File Comments

At the beginning of the file there is a prelude which Sun WorkShop Visual reads and, effectively, throws away. The prelude is always regenerated anew. Before every stub Sun WorkShop Visual generates a comment giving the name of the callback or method. In this way Sun WorkShop Visual can calculate which stubs it still needs to generate, having read the existing stubs file.

Note – You should not alter these comments in any way unless you wish Sun WorkShop Visual to regenerate the stub.

Regeneration of Callback Stubs

If you wish Sun WorkShop Visual to regenerate one of the stubs, simply remove the comment preceding the stub and the stub itself. If you remove only one or the other, one of the following will occur:

- If you remove the stub but leave the comment, no stub will be generated
- If you remove the comment but leave the stub, you will have two copies of the stub

Remember that regeneration of a stub will lose the contents of the routine.

Sun WorkShop Visual will not remove old stubs even though a special comment no longer matches a callback or callback method.

Regeneration of the Whole File

You may wish Sun WorkShop Visual to regenerate the whole file anew if, for example, you have deleted some callbacks or changed some names, the old ones are still being generated and the file is starting to become full of redundant code. In order to do this, simply remove, or change the name of, the stubs file and the "`.bak`" file. If Sun WorkShop Visual cannot find a file with the name you have specified for the stubs file, it will generate a new file.

Analysis of the Primary Module

From top to bottom, your primary code module contains the following sections:

- Headers (optional)
- Global variable declarations
- Declarations of link functions, or the functions themselves (optional)
- Structures needed for your fonts and pixmaps and code setting up font and pixmap objects
- Widget creation code for your design hierarchy

- A *main()* procedure (optional)

This section analyzes the code in your file.

● **Open `icecream.c` with the text editor and inspect it as you read.**

The optional portions of the file can be included or excluded by setting toggles on the control panel. These toggles, and the advantages and disadvantages of including the optional sections, are discussed in "Code Generation Options" on page 218.

The Header Section

The primary module has the following header material, in this order:

- The module heading (if any)
- Sun WorkShop Visual's heading
- The *#include* statements (optional)
- The module prelude (if any)

Your file does not need to have a module heading or module prelude. You can specify code to be inserted in these places from within Sun WorkShop Visual. The procedure for doing so is discussed in "Customizing the Generated Files: Preludes" on page 239.

After some standard Sun WorkShop Visual comments, there is a list of *#include* statements needed for the code in your module. The *#include* statements are optional and are controlled by the toggles in the Code Options dialog - see "Setting up the Primary Source File" on page 211.

Sun WorkShop Visual also needs to include its own header file in order to define the base classes that it uses. If you wish to change the name of the file to be included, or not include a base class header file at all, refer to "Generation" on page 875 for details of the application resource that you will need to change.

Link Functions or Link Declarations

Next, the module contains code for the link functions. The following code fragment shows a typical link function:

```
void XDunmanage_link ( Widget w, XtPointer client_data, XtPointer
call_data )
```

Generation of this code is optional and is controlled by the Generate Options dialog - see "Code Generation Options" on page 218.

Variable Declarations

In this section, all globally defined widgets in the design are declared. The following lines are typical:

```
Widget exit_button = (Widget) NULL;
Widget help_cascade = (Widget) NULL;
```

Only global widgets are declared here. By default, widgets are local in scope. Local widgets are defined in the function which creates their parent Shell and cannot be referenced elsewhere in your application. To make a widget global, you can:

- Specify it as global on the Core resource panel
- Give it an explicit variable name

Note that the variable names of Application Shells and Top level Shells are always global in Sun WorkShop Visual and therefore should not be made local. See "Shell Types" on page 73 for more information on the different shell types.

Variable Names

Variable names must be unique. If you "Read" or "Paste" widgets into your design whose variable names duplicate names of existing widgets, Sun WorkShop Visual silently removes the duplicate names and assigns new, local, names of the form *widget_type<n>*, e.g. *shell4*, *form5*, etc.

By convention, variable names of widgets should begin with a lower-case letter. This helps avoid conflict with Motif declarations.

Creation Procedures

By default, Sun WorkShop Visual generates a *creation procedure* for each Shell widget in your design. The creation procedures are the heart of the generated code. Each creation procedure does the following:

- Creates the Shell widget itself
- Creates and manages all the children of the Shell and their children
- Sets all hard-wired resources for any child of the Shell
- Adds callbacks and (optionally) links to any child of the Shell which has them

The creation procedures do not display the Shell. Usually, windows are displayed by a function call in the *main()* procedure or in a callback routine.

By default, creation procedures have the form *create_<shell name>*, based on the variable name of the Shell. Your design has two Shells: a Dialog Shell, named *help_window*, and an Application Shell, named *myFirstShell*. It therefore has two creation procedures: *create_myFirstShell* and *create_help_window*. You can change the name of a creation procedure in a code prelude, discussed later in this chapter.

create_help_window has the following form:

```
void create_help_window (Widget parent)
{
    . . .
}
```

The function body has function calls which create the Dialog Shell itself:

```
help_window = XmCreateDialogShell ( parent, "help_window", al, ac );
```

Dialog Shells, unlike Application Shells, are dependent on another Shell, *parent*. See “Shell Types” on page 73 for more details concerning the various Shell types and their respective behavior.

create_help_window also creates all the Shell’s children. The DialogTemplate child, to which you gave an explicit variable name, is created and assigned to a global variable:

```
dialog_2 = XmCreateMessageBox ( help_window, "dialog_2", al, ac);
```

The Label, if you did not name it explicitly, is assigned to a local variable as illustrated below. (Note that the widget variable name may be different in your code.)

```
label1 = XmCreateLabel ( dialog_2, "label1",al,ac);
```

create_myFirstShell, the creation procedure for your Application Shell, has different arguments because it is a different type of Shell:

```
void create_shell_1 (Display *display, char *app_name, int app_argc,
char **app_argv)
{
. . .
}
```

See “Shell Pre-create Prelude” on page 244 for a discussion of these arguments.

This function is similar to *create_help_window*, although it is considerably longer as your main window has more child widgets than the help window.

Callback Procedures

The primary module does not include callback functions themselves. However, it does add any callbacks you have specified to each widget's callback list. *create_myFirstShell* contains the following lines (not necessarily together) which create the *exit_button* and add the *quit* callback.

```
exit_button = XmCreatePushButton ( rowcol1, "exit_button", al, ac );
. . .
XtAddCallback (exit_button, XmNactivateCallback, quit, NULL);
```

An extern declaration of *quit()* is generated earlier in the source file.

The "Show" link on the widget *help_button* inserts an Activate callback to the Sun WorkShop Visual function *XDmanage_link*. The code which creates *help_button* and adds a link to it looks much like the code which creates *exit_button* and adds its callback.

```
help_button = XmCreatePushButton ( rowcol2, "help_button", al, ac );
. . .
XtAddCallback (help_button,XmNactivateCallback, XDmanage_link,
(XtPointer) &xd_links[0] );
```

Description of the Main Program

A minimal *main()* procedure is either generated into a separate file or at the end of your primary module. See "Setting up the Main Program File" on page 214 for details on generating this procedure into a separate file.

Sun WorkShop Visual's *main()* procedure does the following things:

- Opens a connection to the X server
- Initializes the X toolkit
- Calls the creation procedure for the first Application Shell
- Calls the creation procedures for all other Shells in the design
- Calls *XtRealizeWidget()* to display the first Application Shell
- Calls *XtAppMainLoop() (*which never returns)
- Calls *exit() (*This call is for neatness only, since this line of code is never executed)

As you have seen, this *main()* procedure is sufficient to run the interface and check its behavior. In many applications, very little additional code is needed in *main()* because most functionality is handled in callbacks. However, if you need to initialize other parts of your application, you should generate a separate *main()* procedure source file from the Generate dialog once only and edit the file. Termination code goes in the callback function which is invoked to exit the application.

● **Close the file `icecream.c`.**

Resource File Syntax

The syntax for generated resource files, by default, is as follows:

<application name>*<widget name>.<resource>: <value>

For identification purposes, the widget’s variable name (not the widget name) precedes the list of its resources in a comment. If a group of widgets share a widget name, however, only one variable name from the group appears. A comment is also generated for any widget which has no resources generated into the file.

Example Syntax

- **Open your X resource file (icecream.res) with a text editor and look at it.**

The file fragment below includes only String resources.

```
! button1
XDTutorial*button1.labelString: Cone

! button2
XDTutorial*button2.labelString: Dish

! button3
XDTutorial*button3.labelString: Cancel

! procedure_cascade
XDTutorial*procedure_cascade.labelString: Procedures

! button4
XDTutorial*button4.labelString: Wash Dishes...

! button5
XDTutorial*button5.labelString: Count Money

! exit_button
XDTutorial*exit_button.labelString: Exit
XDTutorial*exit_button.accelerator: Ctrl<Key>E
```

```
XDTutorial*exit_button.acceleratorText: Control + E

! help_cascade
XDTutorial*help_cascade.labelString: Help
XDTutorial*help_cascade.mnemonic: H

! help_button
XDTutorial*help_button.labelString: About This Layout
XDTutorial*help_button.mnemonic: A
```

An end user can change any of these strings by editing its value in the X resource file. For example, the second line could be changed to read:

```
XDTutorial*procedure_cascade.labelString: Closing Up
```

Resource values in the X resource file are overridden by values for the same resource in the *.Xdefaults* file in the user's home directory.

Include in Resource Binding

If you have set any of the "Include in resource binding" toggles (found on the "Code generation" page of the Core Resource panel) for any widgets in your design, your resource file will look slightly different. The syntax that Sun WorkShop Visual generates for resource files by default is very general - it applies to *all* widgets with the specified widget name within the whole application. It is often the case that you have more than one widget with the same name. Tight bindings give you more control over widget resources.

Note – This section only discusses the syntax of the generated resource file, refer to "Tight Bindings" on page 92 for a thorough explanation of tight bindings.

The example given in "Tight Bindings" on page 92 would produce the following line:

```
XDTutorial*FirstForm*OkButton.labelString: Ok
```

Loose Bindings

If you have set any loose bindings, these will appear at the top of the generated resource file. Their syntax is slightly different as they never refer to individual widgets.

Note – This section only describes the syntax of resource files - for a thorough explanation of loose bindings, refer to "Loose Bindings" on page 86.

The example given in "Loose Bindings" on page 86 would generate the following line in the resource file:

```
XDTutorial*XmDialogShell*MyForm.MyButton.labelString: Bound
```

Shared Resource Values

To identify a widget completely, X requires a list of all the widget's ancestors in the hierarchy as well as the widget's own name. In the generated X resource file, Sun WorkShop Visual uses a wildcard (*) instead of a list of specific ancestors. Thus, each widget is distinguished only by the application name and the widget name, and any widgets which share a widget name, also share any resources generated into the X resource file.

The following lines are taken from Sun WorkShop Visual's own X resource file:

```
/* dialog buttons */
visu*apply_button.labelString: Apply
visu*cancel_button.labelString: Close
```

Sun WorkShop Visual has several buttons, in several places, which have the widget name *apply_button*. All these buttons share the label string "Apply". Similarly, all buttons with the widget name *cancel_button* share the label string "Close". These strings can be changed on all buttons at once by editing one line of the X resource file.

By contrast, resources generated into the source file are always set separately for each widget, even if widgets share a widget name.

Arranging Your Files

Sun WorkShop Visual allows considerable flexibility in arranging files to suit your preference. This flexibility requires some care on your part, since you must include all necessary pieces of code, yet avoid duplication, in order for your application to link successfully.

Another consideration is that your file setup should allow changes to your interface in Sun WorkShop Visual after the first pass at generating code. Remember that any changes you make will require regenerating code and, possibly, resource files. Your files and directories should be set up so that when you regenerate files you do not overwrite any coding work you have done.

With these considerations in mind, this section discusses strategies for organizing your code files.

Using Separate Directories

It is a good practice to keep a separate directory for each Sun WorkShop Visual application. Make the directory before you start designing. Save your design file and generate all code files and resource files into that directory.

Keeping Generated Files Unchanged

To avoid errors, do all of your own coding outside the primary module and X resource file generated by Sun WorkShop Visual. Code preludes, module preludes (see “Customizing the Generated Files: Preludes” on page 239) and the various options in the Generate dialog give you some control over the primary module from within Sun WorkShop Visual. Similarly, resource preludes let you adjust the X resource file. If you do not edit these files outside Sun WorkShop Visual, you can regenerate them when you make changes in your design without sacrificing any work you have done.

Keeping Main Separate

The *main()* procedure almost always needs to be edited. For this reason it is best to generate a separate main program file. You can then edit this file as you wish. Make sure that you do not regenerate the main program file once you have made your own changes to it. See “Setting up the Main Program File” on page 214 for details on how to generate a separate main program file.

Stubs File

Unlike other generated files, the stubs file is meant to be edited. Sun WorkShop Visual will preserve changes to stubs files on re-generation. See Chapter 6 ”Activating the Interface: Adding Your Own Code“ for details on adding, editing and understanding callbacks. See “Adding Callback Functionality” on page 226 for details on editing a stubs file.

Where to Put Links

If your application uses links, you must generate the link functions and function declarations into either the code file or the stubs file.

If your application uses generated code from more than one design file, you should generate the link functions declarations into all the primary modules but generate the link functions into only one file.

Where to Put Includes

If your *make* procedure involves compiling the primary module and the application code separately, you should turn on the “Include Header File” toggle for the primary source code file. This procedure was followed in the tutorial. See “Setting up the Primary Source File” on page 211 for details on how to do this.

Another strategy involves writing a *#include* directive to include the generated code in your application code file and compile all the code together. If you do this, you should turn on “Include Header File” only once for the primary module and turn it off when you generate the stubs file.

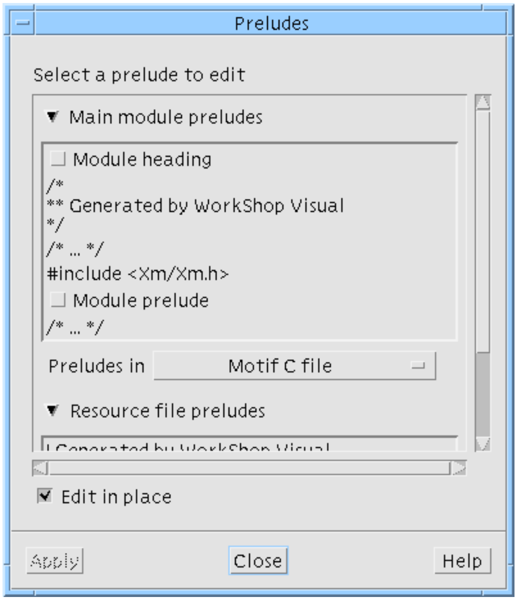
Customizing the Generated Files: Preludes

Sun WorkShop Visual lets you edit the primary source file, which it generates, in order to add lines of your own code, here called *preludes*. All preludes can be entered by typing them into Sun WorkShop Visual and allowing Sun WorkShop Visual to insert them into the code at the appropriate place. There are several types of prelude, distinguished by where the code is inserted.

Note – After adding any type of prelude, you should re-generate code in order to see your changes.

Module Preludes

The “Module prelude...” command in the “Module” menu lets you enter a *heading prelude*, a *module prelude* and a *resource prelude* in your code, using the dialog shown in Figure 7-8.



□ Module heading

```
/*
** Generated by WorkShop Visual
*/
/* ... */
#include <Xm/Xm.h>
```

□ Module prelude

```
/* ... */
```

Preludes in: Motif C file

▼ Resource file preludes

```
! Generated by WorkShop Visual
```

☑ Edit in place

FIGURE 7-8 Module Prelude Dialog

Selecting one of the toggles which appear in the representative text, allows you to edit that type of prelude. There are two ways of adding a code prelude:

- Editing the generated code
- Typing the code into the dialog

Use the “Edit in place” toggle to specify which of the above you wish to use. If the “Edit in place” toggle is set, the generated code is opened for you to add your code. See “Using the Edit Mechanism” on page 246 for more details on this.

If the “Edit in place” toggle is not set, a large text widget appears on the right of the dialog. Enter your code here. You should type the code exactly as if you were using a text editor to type any other code. This means that you should observe all the rules and conventions of the target language, including end of line markers, bracketing conventions etc. You should always press Return after the last line in a prelude.

Heading Prelude

The heading prelude is inserted at the beginning of the main program file, the code file, the externs file and the stubs file. Typically, a module heading would contain a comment with information such as the program name, SCCS ID, or version number.

Module Prelude

The module prelude is inserted at the top of the generated code file - just after Sun WorkShop Visual’s generated *#include* statements, if you asked for them. The module prelude can be used to supply *#define* or *#include* statements or *extern* declarations which are needed by your code. The module prelude is generated only into the primary module, not the stubs file.

Resource Prelude

The resource prelude is inserted at the beginning of the X resource file to specify application resources - i.e. resources which refer to the whole application and not to individual widgets. Use the following syntax:

```
ApplicationName*resource: value
```

For example:

```
visu*symbolFont: -*-symbol-medium-r-normal--14*
```

Although these resource bindings apply to all widgets in the application, they are overridden by more specific resource settings on individual widgets or groups of widgets with a common name.

You may also wish to include comments or SCCS or RCS keywords in a resource prelude.

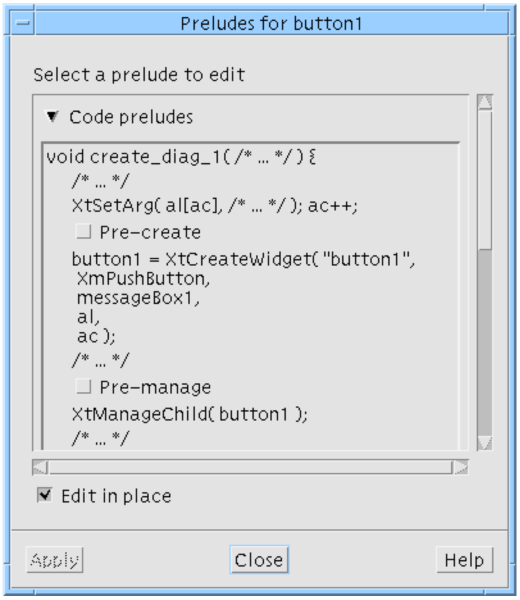
See “Loose Bindings” on page 86 for information on setting up loose resource bindings for the whole module and for individual widgets.

Code Preludes

While module preludes apply to the whole module, code preludes apply to individual widgets. This means that the code will be inserted before the widget is created or managed. To add prelude code:

1. **Select the widget to which you wish to add code**
2. **Select "Code Preludes" from the "Widget" menu**

The Code Prelude dialog is shown in Figure 7-9:



Preludes for button1

Select a prelude to edit

▼ Code preludes

```
void create_diag_1( /* ... */ ) {
   /* ... */
   XtSetArg( al[ac], /* ... */ ); ac++;
    □ Pre-create
   button1 = XtCreateWidget( "button1",
    XmPushButton,
    messageBox1,
    al,
    ac );
   /* ... */
    □ Pre-manage
   XtManageChild( button1 );
   /* ... */
```

☑ Edit in place

FIGURE 7-9 Code Preludes Dialog

Code Preludes Dialog

The Code Prelude dialog contains text representing generated code for the selected widget. This code is representative only, so that you can see where the preludes will be added. This is not the actual generated code.

This dialog contains two sections - one for C code (labelled "Code preludes") and one for C++ (labelled "Method preludes"). Within both sections of text are toggles allowing you to choose whether you wish to edit the various kinds of prelude. There are two kinds of code prelude which can be used with C: *Pre-create* and *Pre-manage*. These are discussed in “Pre-create Preludes” on page 243 and “Pre-manage Preludes” on page 245.

There are three kinds of method preludes for use with C++: *public*, *private* and *protected*. These relate to their access. “Method Access Control” on page 260 provides more information on method access. You can, however, add both methods and data members in a prelude. “Adding Class Members as a Prelude” on page 291 shows, as part of a tutorial, how to add data members using the Code Prelude dialog.

Clicking over “Code preludes” or “Method preludes” folds away the corresponding text area so that only the other prelude type is visible.

Adding a Code Prelude

Selecting one of the toggles which appear in the representative text, allows you to edit that type of prelude. There are two ways of adding a code prelude:

- Editing the generated code
- Typing the code into the dialog

Use the “Edit in place” toggle to specify which of the above you wish to use. If the “Edit in place” toggle is set, the generated code is opened for you to add your code. See “Using the Edit Mechanism” on page 246 for more details on this.

If the “Edit in place” toggle is not set, a large text widget appears on the right of the dialog. Enter your code here. You should type the pre-creation code exactly as if you were using a text editor to type any other code. This means that you should observe all the rules and conventions of the target language, including end of line markers, bracketing conventions etc.

Pre-create Preludes

Pre-create preludes are inserted into the code before the selected widget is created.

If the selected widget is not a Shell widget, the pre-creation prelude is inserted in the creation procedure for the widget’s parent Shell and you can provide any code without restriction. Pre-creation preludes are commonly used to set resources which can only be set at widget creation time. Pre-create preludes for Shells are different and are described in “Shell Pre-create Prelude” on page 244. Below is a segment of generated code showing where pre-create preludes are added:

```
          ...
/* visu: prelude for rowcol1: pre-create >>> */
    Enter pre-create code here
```

```
/* <<< pre-create ends. */
rowcol1 = XmCreateRowColumn ( shell11, “rowcol1”, al, ac );
          ...
```

If you had selected “Edit in place” and you are editing the generated code directly, make sure that your code is typed into the area surrounded by special comments. It is then preserved when you regenerate code. You must not alter or remove the special comments.

Shell Pre-create Prelude

The code preludes for a Shell differ slightly from those of other widgets. The pre-creation prelude is used to *replace* the function header for the Shell’s creation procedure. You can then, if you wish, define extra parameters.

The generated body of the procedure refers to one or more variables, which, in the default procedure heading, are passed as parameters. While these variables must be in scope, you can choose to pass them as parameters or declare them as global variables. The following variables must be in scope:

Required for Application Shell widgets:

```
Display *display;
char *app_name;
int app_argc;
char **app_argv;
```

Required for Dialog Shell or Top level Shell widgets:

```
Widget parent;
```

In C for UIL the following are also required:

```
MrmHierarchy hierarchy_id;
MrmCode *class;
```

If you do not provide a pre-create prelude for a Shell widget, the creation procedure name defaults to *create_<widget-variable-name>* with the compulsory parameters as the only parameters.

Note – If you provide a pre-create prelude for a Shell, the call of the creation procedure in the generated default *main()* program is unlikely to be correct.

WARNING – Shell pre-create preludes cannot be edited in place for Motif XP or MFC code. In general, code preludes should not be used for cross-platform designs because, by their very nature, preludes tend to include platform-specific code.

Pre-manage Preludes

The pre-manage prelude appears slightly later than the pre-create prelude in the generated code - just before the widget's callbacks are added. One use of this prelude is to set up client data for the callbacks. Other uses include setting the value of a Text widget, filling a ScrollingList, adding buttons from a file, or any other dynamic initializations. Below is a segment of generated code showing where pre-manage preludes appear:

```
          ...
/* visu: prelude for shell1: pre-manage >>> */
     Enter pre-manage code here
/* <<< pre-manage ends. */
XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
XtSetArg(al[ac], XmNargc, app_argc); ac++;
XtSetArg(al[ac], XmNargv, app_argv); ac++;
          ...
```

If you had selected “Edit in place” and you are editing the generated code directly, make sure that your code is typed into the area surrounded by special comments. It is then preserved when you regenerate code. You must not alter or remove the special comments.

Shell Pre-manage Prelude

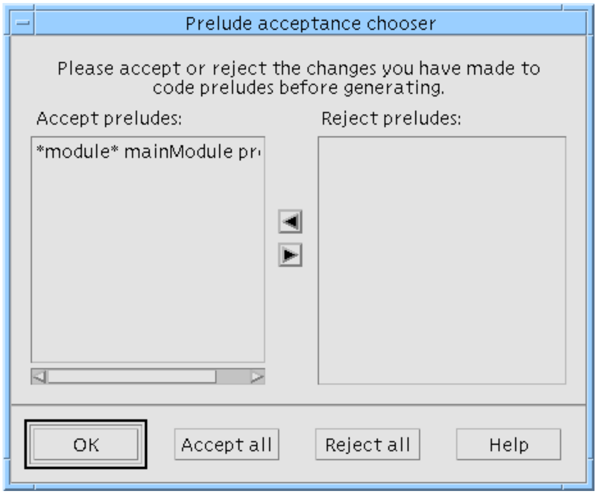
A Shell's pre-manage prelude is inserted just after the local declarations in the procedure. Otherwise it is the same as for other widgets.

Using the Edit Mechanism

Sun WorkShop Visual uses the SunSoft Workshop Edit Server when you edit the generated code file. The file is opened in a separate editing window at the appropriate place in the file for the selected prelude type. As you select other prelude types the insertion point moves around the file so that you are always at the correct place. If you try to regenerate the code file when there are unsaved changes in the Edit Server window, you will be prompted to save the changes first.

Prelude Acceptance Chooser

When you regenerate the code file after having added a prelude in place, the Prelude Acceptance Chooser is displayed. This is shown in Figure 7-10. Adding a prelude directly into the Preludes dialog will not cause the Prelude Acceptance Chooser to appear.



Please accept or reject the changes you have made to code preludes before generating.

module mainModule pr

FIGURE 7-10 Prelude Acceptance Chooser

In this dialog, you can choose individual preludes which have changed since the file was last generated and specify whether you wish to accept or reject those preludes. Select the preludes and press the arrow keys to move them from one list to the other. When you press "Ok" only those preludes you chose to accept will be regenerated. Alternatively, you can choose to reject or accept *all* the newly added preludes. Preludes which you reject are deleted when you "Ok" the dialog and cannot be retrieved at a later date.

Preludes which were added before a previous code generation took place are retained - the Prelude Acceptance Chooser only affects those preludes which have been added since your code file was last generated.

Note – If you unset the “Edit in place” toggle and you have added a prelude since the last code file generation the Prelude Acceptance Chooser is displayed because any new preludes need to be shown in the Preludes dialog.

CHAPTER 8

Structured Code Generation and Reusable Definitions

Introduction

This chapter describes how Sun WorkShop Visual helps you to control the structure of the generated code. Being able to do this is essential for creating reusable widget hierarchies. These reusable hierarchies, known as *definitions*, appear on the widget palette and can be added to the hierarchy like any other widget. A detailed description of definitions also appears in this chapter.

Structured Code Generation

Sun WorkShop Visual provides controls for structuring your generated code so that it is more flexible and can be reused more easily. Before reading this section, you should review the structure of the default generated code in “Analysis of the Primary Module” on page 230. In particular, note that the default code has a single creation procedure for each Shell in the design. Widgets are declared as local if they have not been named and global if they are named or are Application Shells.

The structured code controls let you:

- Designate any widget in the hierarchy to have its own creation function that returns the widget, including its descendants
- Designate any widget to have its own creation function that returns a structure containing the widget and its named descendants

- Designate any widget to be defined as a C++ class with descendant widgets as members
- Designate a widget as a place-holding container that serves only to house a collection of child widgets
- Explicitly specify a widget as global, local, or static

Sun WorkShop Visual's controls for structuring code are located on the "Code generation" page of the Core resource panel.

Function Structures

The simplest case of structured code generation is to designate a widget as a *function structure*. This makes Sun WorkShop Visual generate a separate function that creates that widget and its descendants. This function is called by the creation procedure for the enclosing widget.

To do this, select the "Code generation" page of the Core resource panel and select "Function" from the "Structure" option menu.

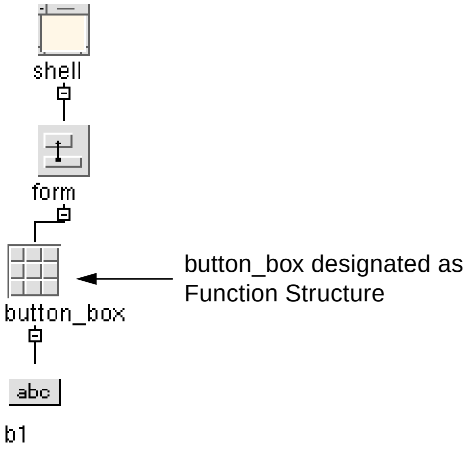


FIGURE 8-1 Example: Structure

The hierarchy shown in Figure 8-1 produces the following generated code, slightly simplified for clarity:[1]

```
Widget shell = (Widget) NULL;
Widget form = (Widget) NULL;
```

[1] The comments describing the functions and procedures are not generated.

```
Widget button_box = (Widget) NULL;
Widget b1 = (Widget) NULL;
/* This is the creation function for the button_box. */
Widget create_button_box (Widget parent)
{
    Widget children[1];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */
    Widget button_box = (Widget)NULL;

    button_box = XmCreateRowColumn ( parent, "button_box",
          al, ac );
    b1 = XmCreatePushButton ( button_box, "b1", al, ac );
    children[ac++] = b1;
    XtManageChildren(children, ac);
/* The button box is created, but not managed, and returned. */
    return button_box;
}
/* The creation function for the Shell calls the button box creation function. */
void create_shell (Widget parent)
{
    Widget children[1];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    shell = XmCreateDialogShell ( parent, "shell", al, ac );
    ac = 0;
    XtSetArg(al[ac], XmNautoUnmanage, FALSE); ac++;
    form = XmCreateForm ( shell, "form", al, ac );
    ac = 0;
    button_box = create_button_box ( form );
/* The constraint resources for the button box are set in the parent’s creation
function. */
    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM);
    ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM);
    ac++;
    XtSetValues ( button_box,al, ac );
/* The button box is managed at this point. */
    children[ac++] = button_box;
    XtManageChildren(children, ac);
}
```

This module now has two functions: one (*create_shell()*) for creating the whole hierarchy and one (*create_button_box()*) for creating the button box.

Data Structures

The next type of code structuring is the *data structure*. This is similar to a function structure, in that Sun WorkShop Visual generates a separate creation procedure for the widget and its descendants. When a widget is designated as a data structure, Sun WorkShop Visual also generates a *typedef* for a structure including that widget and its children. The creation procedure for the widget creates and sets up that type of structure and returns a pointer to it. A deletion function (*delete_<widget_name>*) is also generated so that the allocated memory can be freed.

To designate a widget as a data structure, select the “Code generation” page from the Core resource panel and select “Data structure” from the “Structure” option menu.

Using the same hierarchy as shown above, but with *button_box* designated as a data structure, the following code is produced, slightly simplified for clarity:[1]

```
/* First the type declarations are generated for the data structure. */
typedef struct button_box_s {
    Widget button_box;
    Widget b1;
} button_box_t, *button_box_p;
Widget shell = (Widget) NULL;
Widget form = (Widget) NULL;
button_box_p button_box = (button_box_p) NULL;
/* The creation procedure returns a pointer to a button_box structure. */
```

[1] The comments describing the functions and procedures are not generated.

```
button_box_p create_button_box (Widget parent)
{
    Widget children[1];      /* Children to manage */
    button_box_p button_box = (button_box_p)NULL;
/* Space is allocated for the structure and the fields are filled in. */
    button_box = (button_box_p) XtMalloc ( sizeof (
          button_box_t ) );
    button_box->button_box = XmCreateRowColumn ( parent,
          "button_box", al, ac );
    button_box->b1 = XmCreatePushButton
          ( button_box->button_box, "b1", al, ac );
    children[ac++] = button_box->b1;
    XtManageChildren(children, ac);
    return button_box;
}
/* A deletion function is supplied to free the allocated memory. */
void delete_button_box ( button_box_p button_box )
{
    if ( ! button_box )
          return;
    XtFree ( ( char * )button_box );
}
/* Again, the Shell creation function calls the button box creation function. */
void create_shell (Widget parent)
{
    Widget children[1]; /* Children to manage */
    Arg al[64]; /* Arg List */
    register int ac = 0; /* Arg Count */
    shell = XmCreateDialogShell ( parent, "shell", al, ac );
    form = XmCreateForm ( shell, "form", al, ac );
    button_box = create_button_box ( form );
    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM);
    ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM);
    ac++;
/* The button_box widget has to be referenced inside the structure. */
    XtSetValues ( button_box->button_box,al, ac );
    ac = 0;
    children[ac++] = button_box->button_box;
    XtManageChildren(children, ac);
    ac = 0;
}
```

C++ Classes

The use of *C++ classes* is very similar to data structures. Sun WorkShop Visual does not wrap each widget in the hierarchy with a C++ class, but instead designates sections of the hierarchy as classes in their own right. Each widget designated as a C++ class has a class defined for it. Its named descendant widgets become members of that class and widget creation and widget destruction methods are supplied. In addition, if the class contains members that are themselves (pointers to) classes, a constructor and destructor method is generated to create and destroy these members. Note that the widgets are not created at the time of the class instance but by an explicit call to the widget creation function. Similarly, destroying the class instance does not destroy the widgets.

To designate a widget as a C++ class, select the “Code generation” page of the Core resource panel and select “C++/Java class” from the “Structure” option menu. Note that if you designate a widget as a C++ class, then generate C, the widget is treated as a data structure.

This section describes C++ classes. For information Java classes in Sun WorkShop Visual, see Chapter 10, “Designing for Java”, starting on page 313.

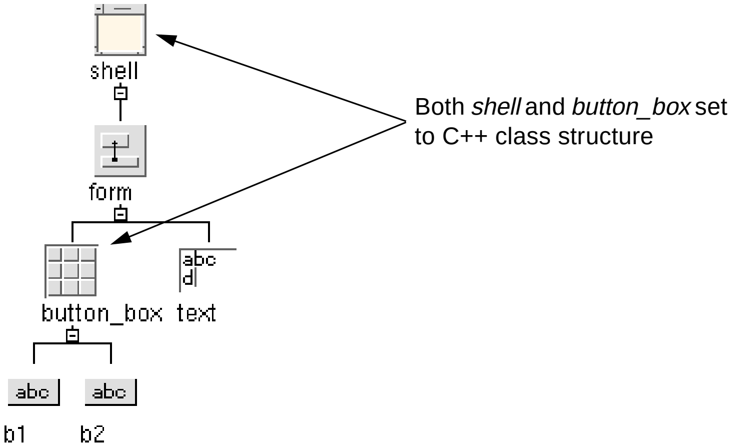


FIGURE 8-2 Example: C++ Class Structures

The C++ code generated from this example is shown below, simplified for clarity:[1]

Classes are declared for *button_box* and *shell*:

```
class button_box_c: public xd_XmRowColumn_c {
public:
    virtual void create (Widget parent, char *widget_name =
         NULL);
protected:
    Widget button_box;
    Widget b1;
    Widget b2;
};

typedef button_box_c *button_box_p;
```

The *shell* class has constructor and destructor functions because it contains a pointer to class (or data structure) member:

```
class shell_c: public xd_XmDialog_c {
public:
    virtual void create (Widget parent, char *widget_name =
         NULL);
    shell_c();
    virtual ~shell_c();
```

[1] The comments describing the functions and procedures are not generated.

```
protected:
    Widget shell;
    Widget form;
    Widget text;
    button_box_p button_box;
};

typedef shell_c *shell_p;

shell_p shell = (shell_p) NULL;
```

The creation function now becomes a method of the class. This method is declared public in the Sun WorkShop Visual base class, which is supplied with the release:

```
void button_box_c::create (Widget parent, char *widget_name)
{
    Widget children[2];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    if ( !widget_name )
          widget_name = "button_box";

    button_box = XmCreateRowColumn ( parent, widget_name,
          al, ac );
```

_xd_rootwidget is a protected member of the class that stores the widget that is at the root of the sub-hierarchy. This lets the base class operate on the widget:

```
    _xd_rootwidget = button_box;
    b1 = XmCreatePushButton ( button_box, "b1", al, ac );
    b2 = XmCreatePushButton ( button_box, "b2", al, ac );
    children[ac++] = b1;
    children[ac++] = b2;
    XtManageChildren(children, ac);
    ac = 0;
}
```

// The Shell’s creation method calls that for the button box.

```
void shell_c::create (Widget parent, char *widget_name)
```

```
{
    Widget children[2];      /* Children to manage */
    Arg al[64];                    /* Arg List */
    register int ac = 0;           /* Arg Count */

    if ( !widget_name )
         widget_name = "shell";

    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    shell = XmCreateDialogShell ( parent, widget_name, al,
         ac );
    ac = 0;
    _xd_rootwidget = shell;
    XtSetArg(al[ac], XmNautoUnmanage, FALSE); ac++;
    form = XmCreateForm ( shell, "form", al, ac );
    ac = 0;
    text = XmCreateText ( form, "text", al, ac );
```

The button box class is instantiated in the constructor method and so at this point only the widgets need to be created:

```
    button_box->create ( form, "button_box" );

    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET);
    ac++;
    XtSetArg(al[ac], XmNtopWidget,
         button_box->xd_rootwidget()); ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM);
    ac++;
    XtSetValues ( text,al, ac );
    ac = 0;

    XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM);
    ac++;
    XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM);
    ac++;
    XtSetValues ( button_box->xd_rootwidget(),al, ac );
    ac = 0;
    children[ac++] = text;
    children[ac++] = button_box->xd_rootwidget();
    XtManageChildren(children, ac);
    ac = 0;
}

shell_c::shell_c()
{
```

Instantiate the child classes:

```
    button_box = new button_box_c;
}
shell_c::~shell_c()
{
```

Free the child classes:

```
    delete button_box;
}
```

If a widget is designated a C++ class and C code is generated, the widget is treated as if it were a data structure.

By default, the generated class is derived from one of the supplied Sun WorkShop Visual base classes. You can override this by specifying the base class in the field below the C++ Access option menu. The Sun WorkShop Visual base classes supplied with the release provide minimal support, sufficient for the generated code to execute correctly. You can modify and extend those classes to provide reusable methods that suit your approach to GUI development.

Descendant widgets appear as protected members of the class if they are named, or if they are themselves data structures or C++ classes. It is therefore important to name the C++ class widget itself and any of its descendants that you want to access as class members. You can alter the default access control by selecting the required level (Public, Protected, or Private) from the C++ Access option menu.

Using an unnamed widget for the C++ class widget itself does not cause an immediate error. However, this is not recommended as numbers assigned by Sun WorkShop Visual can change when you edit your hierarchy.

Callback Methods

The X toolkit functions which invoke callback functions expect a callback function in the following form:

```
void my_callback (Widget, XtPointer, XtPointer)
```

An ordinary member function is not suitable as a callback function because the C++ compiler passes it an extra first parameter - the *this* pointer - that lets it find the instance data for the object. If you use an ordinary member function as a callback function, the member function interprets the widget pointer as the instance data pointer and does not work as expected.

Sun WorkShop Visual uses a common technique to work around this. A static member function (which does not expect a *this* pointer) is declared and used as the callback function:

```
static void my_callback (Widget, XtPointer client_data, XtPointer
call_data)
```

The client data parameter is used to pass in a pointer to the instance. The static member function merely calls an ordinary non-static member function using that instance pointer and passes on the widget and call data parameters. The non-static member function has the following form:

```
virtual void my_callback (Widget, XtPointer call_data)
```

Sun WorkShop Visual generates both function declarations, all the code for the static callback function and a stub for the regular member function which is written by you. Note, because this function is declared as *virtual*, you can override it in a derived class to modify the behavior. For a discussion of this technique, see "*Object-Oriented Programming with C++ and OSF/Motif*" by Douglas Young.

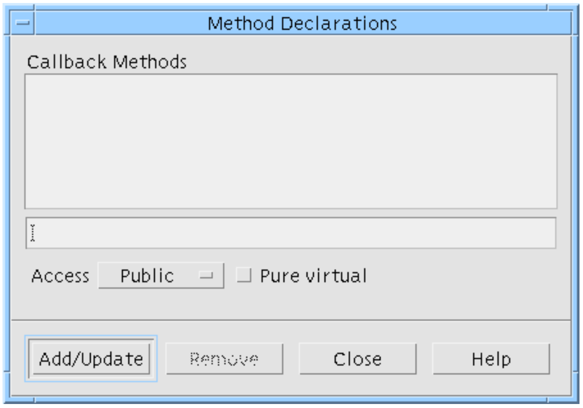
Editing Callback Methods

When you add a callback method, Sun WorkShop Visual also adds a declaration for the method (if it has not already been declared). Pressing the "Methods" button in the Callbacks dialog shows you a list of the methods declared in the enclosing class of the currently selected widget.

By default, Sun WorkShop Visual declares methods as not pure virtual and with public access. If these attributes are not as you intended, use the Method Declarations dialog to change them. See "Method Declarations" on page 260 for details.

Method Declarations

If you add a callback as a method, for convenience Sun WorkShop Visual adds the declaration of the method in the *enclosing class* for that widget. You can view, add and remove method declarations by selecting the widget which is the enclosing class and selecting “Method declarations” from the “Widget” menu. The Method Declarations dialog is shown in Figure 8-3.



Access Public

□ Pure virtual

FIGURE 8-3 Method Declarations Dialog

To find which widget is the enclosing class, use “Structure colors” from the “View” menu, as described in “Structure Colors” on page 47, and select the nearest ancestor of the widget for which you have added a method. Of course, this would be the same widget if it is defined as a C++ class.

Method Access Control

By default, methods added by Sun WorkShop Visual have public access. You can control the access for individual callback methods using the “Access” option menu in the Method Declarations dialog.

Pure Virtual Methods

You can set the “Pure virtual” toggle to declare the non-static member function as pure virtual. For example, if you set this toggle for a callback method *OnNew()* in a menubar class, Sun WorkShop Visual would declare the method as:

```
class menubar_c: public xd_XmMenuBar_c {
...
public:
...
    virtual void OnNew( Widget, XtPointer ) = 0;
};
```

Because the function is pure virtual, you do not have to provide an implementation of *menubar_c::OnNew() and menubar_c* becomes an abstract class. That is, you cannot create an instance of *menubar_c* but only use it as a base class for others.

By default, methods added by Sun WorkShop Visual are *not* pure virtual.

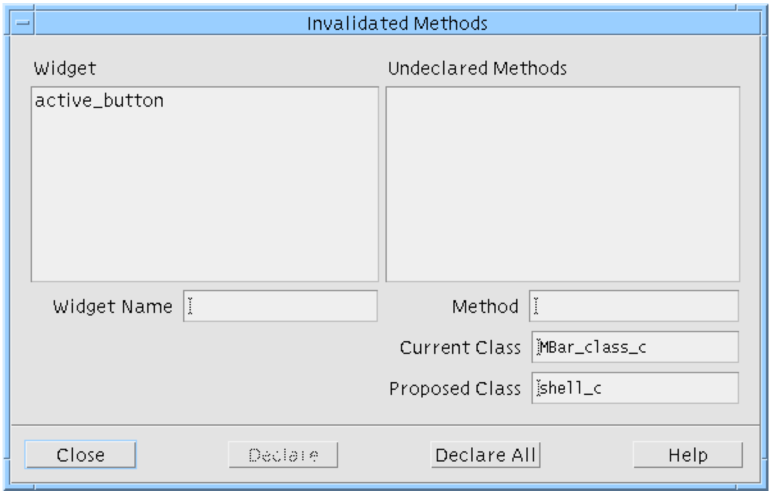
Deleting Callback Methods

When you remove a callback method from a widget you are only removing the *use* of the method (the call to it). When you add a method callback in Sun WorkShop Visual, a declaration of the method is automatically added for you. If you want to remove this declaration as well you must remove it from the method declarations list of the widget which is the enclosing class. See "Method Declarations" on page 260 for more information on how to do this and for information on the declaration added by Sun WorkShop Visual.

Changing the Structure and Invalidating Methods

When a callback method is added, the method is declared in the enclosing class, as described above. If you change the structure of this widget (the enclosing class) so that it is no longer a class, the method becomes invalid. To help you when this happens, Sun WorkShop Visual displays the Invalidated Methods dialog, shown in Figure 8-4.

This dialog is modal - you can not continue working on your design until it is closed. It is only ever displayed when you change a widget's structure in such a way that method declarations are made invalid.



Invalidated Methods

Widget: active_button

Current Class: MBar_class_c

Proposed Class: shell_c

FIGURE 8-4 Invalidated Methods Dialog

The Widget list on the left shows all the widgets with methods which are invalidated by changing the structure. When you select a widget any invalidated methods are listed on the right. For each selected method, this dialog shows you the class in which it is currently declared and suggests a new class for the declaration of your method. The “Proposed Class” is always the nearest ancestor class. If there is no other suitable class, this dialog serves as a warning that the method will become a function.

Pressing “Declare” changes the declaration of the selected method to the “Proposed Class”. Pressing “Declare All” changes each invalidated method to its respective “Proposed Class”.

Method Preludes

You can add additional data or function members to a C++ class using the “Code preludes” dialog. Select “Public methods”, “Protected methods”, or “Private methods” and type your declarations into the text area (or into the code if you are editing in place). C++ code preludes are generated into the class declaration, both in the primary module and in the Externs file.

Creating a Derived Class

To add a function to a class it is often better to write a new class derived from the generated class. The logical gap between the subclass and generated base class can be used to add members and provide implementations for virtual functions.

By default, Sun WorkShop Visual derives the name of a C++ class from the variable name of the root widget and so the class for the widget *menubar* is *menubar_c*:

```
class menubar_c: public xd_XmMenuBar_c {
...
};
```

When Sun WorkShop Visual generates code to create an instance of the class, it uses the same name:

```
menubar = new menubar_c;
```

You can change the default behavior so that Sun WorkShop Visual declares the generated class under one name and creates the instance under another. For example:

```
menubar = new mymenubar_c;
```

To make this change, use the “Instantiate as” field on the Code Generation page of the Core resource panel.

Modifying the Base Classes

By default, Sun WorkShop Visual derives a generated class from a base class appropriate to the type of the root widget. For example, a class with a MenuBar at the root of its widget hierarchy is derived from *xd_XmMenuBar_c*. The name of the base class can be changed in the Core resource panel.

The Sun WorkShop Visual distribution contains a sample implementation of a set of base classes. These can be used as they stand or modified to add extra functionality appropriate to a particular application area.

A Makefile is included to build the sample base classes. Sun WorkShop Visual makes two assumptions about the base classes:

There is a data member *_xd_rootwidget* of type *Widget*.

There is an accessor function *xd_rootwidget()* that returns the value of *_xd_rootwidget* to be retrieved.

These assumptions, together with a few items of basic class restrictions, are encapsulated in the class *xd_base_c*:

```
class xd_base_c
{
public:
    xd_base_c() {_xd_rootwidget=NULL;}
    Widget xd_rootwidget() const {return _xd_rootwidget;}

protected:
    Widget _xd_rootwidget;
private:
    void operator=(xd_base_c&); // No assignment
    xd_base_c(xd_base_c&);      // No default copy
};
```

Sun WorkShop Visual places no other constraints on the base classes used. In other words, any set of base classes can be used provided that they are derived from *xd_base_c* (or another base class that satisfies Sun WorkShop Visual's assumptions).

Note that actual parameters for the base class constructor can be supplied with the class name. If parameters are supplied (if the base class string contains a '()', the class is forced to have a constructor and the parameter string is passed to the base class. For example, setting the "Base class" string to mymenubar_c ("Hello World") for the widget menubar will cause Sun WorkShop Visual to generate:

```
class menubar_c : public mymenubar_c {
public:
    menubar_c();
    ...
};
...
menubar_c::menubar_c () : mymenubar ( "Hello World" )
{
}
...
menubar = new menubar_c;
```

Children Only Place Holders

The *Children Only* structure option lets you designate one widget (the Children Only widget) as a container structure for another structure. Children Only widgets provide context for their descendants in the hierarchy, but no code is generated for them. Consider the following example:

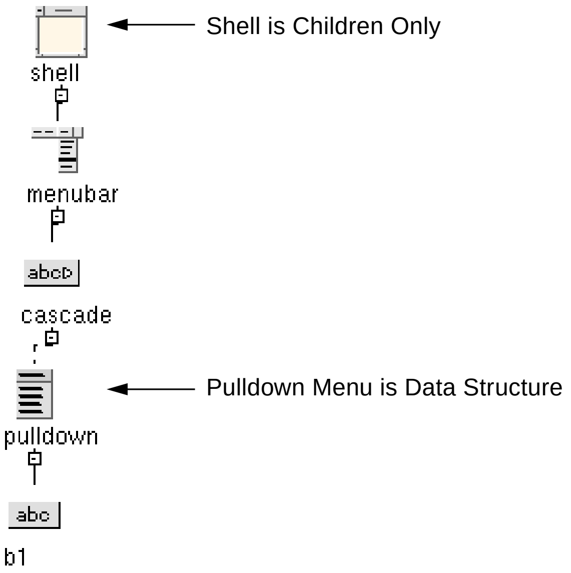


FIGURE 8-5 Use of Children Only Structure

When you generate code from the design shown in Figure 8-5, Sun WorkShop Visual produces code for the pulldown menu structure only. This feature lets you generate fragments of the design that can be controlled by your application program.

Note – If you specify a widget as “children only”, code is only generated for children which are structured or named. Therefore, if all you have underneath a “children only” widget is unstructured and unnamed widgets, then all you will see in the code is Widget declarations.

Children Only Structure in Microsoft Windows Mode

When you are in Microsoft Windows mode, you cannot make the child of a shell a C++ class. To overcome this, so that you can create a hierarchy with a “children only” shell, add a “dummy” container (a rowcolumn or form widget) beneath the shell and then make the container beneath that a C++ class. This would also be useful for creating definitions in Microsoft Windows mode, where the root widget must be structured but the child of the shell cannot be.

Structured Code Generation and UIL

When generating UIL for a design that contains structures of some kind, the approach is basically similar to that for C and C++. Independent hierarchies are generated into the UIL file and separate creation functions are generated into the code file. The creation function fetches the appropriate widgets from the UIL hierarchy and fills in the data structure fields as appropriate.

Changing Declaration Scope

Widgets are normally declared locally in the enclosing creation function unless they are structured in some way, or named. In this case they are declared in the enclosing structure if there is one, or as global variables. This default behavior can be modified by setting the storage class of a widget in the Core resource panel. Setting the storage class to Local forces a widget that would otherwise be declared globally or within a structure to be local to the creation function. Setting the storage class to Global forces an unnamed widget or a named element of a structure to be global. Global status is especially useful for widget-type resources and links as discussed in “Unreachable Widgets” on page 267. The Static option is similar to Global but the declaration is static to the module.

There is no way to force an unnamed widget into a data structure. Unnamed children of a data structure widget are created and managed locally to the data structure’s creation procedure.

Unreachable Widgets

When you use the structured code generation in conjunction with widget-type resources such as *XmNdefaultButton* for a BulletinBoard, you could specify designs that reference widgets that are not in scope. These are considered unreachable widgets. Sun WorkShop Visual attempts to detect these cases and warns you at code generation time. Also, if you use unreachable widgets in conjunction with Children Only structures or dynamic run-time creation of hierarchies, unexpected failures may result.

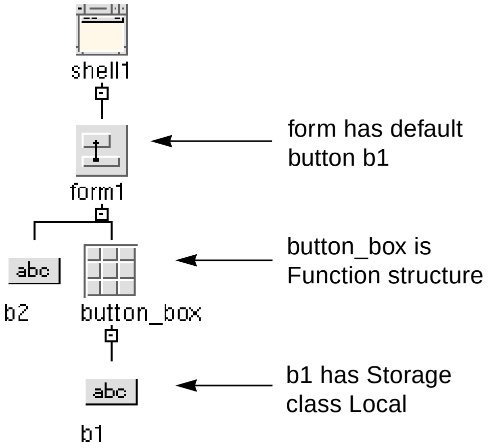
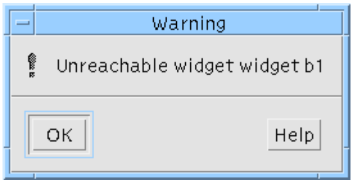

shell1
form1
b2
button_box
b1
form has default button b1
button_box is Function structure
b1 has Storage class Local

FIGURE 8-6 Hierarchy with Unreachable Widgets

An unreachable widget is illustrated in Figure 8-6. *b1* must be available to the Form's creation function so that it can be used as the default button argument. However, since *b1* is local to the *button_box* function, it is not in scope in the Form's creation function. Sun WorkShop Visual detects this situation and displays the following warning at code generation time.



Unreachable widget widget b1

FIGURE 8-7 Unreachable Widget Error

Code is still generated but it may not compile or run as expected. The simplest solution to this is to force the appropriate widget to be global by using the Storage Class option.

Definitions

Once a hierarchy of widgets has been encapsulated as a structure (either a C++ class or a C structure), you can re-use it in other designs by turning it into a *definition*. A definition is a reusable hierarchy of widgets which is added to the Sun WorkShop Visual widget palette. Selecting a definition from the palette creates an instance of the definition in the design. This instance can be further modified and in turn be made into a definition.

Prerequisites

A widget hierarchy can become a definition provided that:

1. The root widget has a non-default variable name.
2. The root widget has been designated as a C++ class or a structure.
3. The root widget is not part of another definition.
4. The widget hierarchy does not contain a definition.
5. The widget hierarchy does not contain any global or static widgets.

Designating a Definition

Designating a definition requires that the design file containing the widget is saved and the widget marked in it as being a definition. To mark the widget as a definition use the Definition toggle in the Widget menu. Creating a definition freezes the widgets within it. Their resource panels are disabled and you cannot add widgets or change widget names. You can edit the widgets that make up a definition only by temporarily removing the definition status. This should be done with caution to avoid conflicts with designs that use the definition. For details, see “Modifying a Definition” on page 272.

To make the definition available for use in other designs Sun WorkShop Visual needs an external reference to it. This is provided by means of a definitions file which is edited using the Edit Definitions dialog.

Definition Shortcut

The “Define” button in the Palette Menu is a quick way of adding a new definition. It designates the currently selected widget as a definition, saves the design and adds the definition to the palette. The header filename for the definition is taken from the type declarations filename in the code generation dialog. No icon is used.

The Definitions File

The definitions file is read by Sun WorkShop Visual to establish the set of definitions which are to appear on the palette. The definitions filename is specified by setting the *definitionsFileName* resource. The default value is *$HOME/.xddefinitionsrc*.

If you need to work on multiple projects, each of which uses a different set of definitions, you can change the definitions file by setting the resource. For example:

```
visu.definitionsFileName:/home/project6/xddefs
```

The value of this resource can include environment variables:

```
visu.definitionsFileName:$PROJECT_ROOT/xddefs
```

To change to the new setting, exit and restart Sun WorkShop Visual.

Editing the Definitions File

To modify the definitions file use the Edit Definitions button in the Palette menu.

This displays the dialog shown in Figure 8-8.

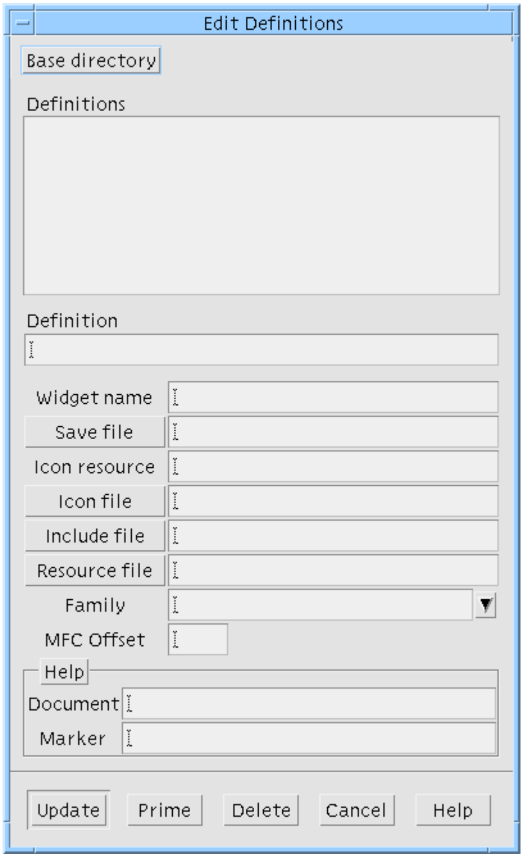


FIGURE 8-8 Adding a Definition to the Palette

You can use this dialog to add a new definition, delete a definition, or edit an existing definition. To add a definition, you must supply:

- *Definition* – A definition name
- *Widget name* – The variable name of the root widget of the definition
- *Save file* – The name of a saved design file (*.xd*)

You can also specify:

- *Icon resource* – A resource name which will be used to locate the pixmap file for the definition. See “Specifying the Icon File” on page 703 for further details
- *Icon file* – A file containing a bitmap or xpm pixmap to be used as the palette icon if one is not found using the Icon resource

- *Include file* – The name of the header file that declares the corresponding structure or class. This file is automatically #included in generated code when instances of the definition are used, therefore you will have to make sure that the compiler can locate it. It must be the same name as the externs file generated from the definition
- *Resource file* – The name of the resource file which contains values for the definition. It is included in the generated resource file when instances of the definition are used. It should correspond to the name specified when the resource file was generated for the definition
- *Family* – The family, or group, to which this definition belongs. This is only relevant to the display of definitions on the widget palette. Definitions are grouped together in families. One family is displayed at any given time. You can change which family is displayed by selecting from the option menu above the definitions on the widget palette. By default, definitions are assigned to the "Default" family. You can specify any name for a family. You can also group any number of definitions in the same family
- *Help information* – A document and tag pair which can be used to provide help to users. See "Online Help for Definitions" on page 276 for more details
- *MFC Offset* – This field is only present when Sun WorkShop Visual is in Microsoft Windows mode. In Microsoft Windows applications controls are given a unique number by which they are identified. Sun WorkShop Visual attempts to generate unique numbers and in most circumstances there will not be a problem. However when adding widgets to an instance which has a very large number of controls already, it is possible for the numbers to overlap. The MFC offset is added to the id of a control which is being added to an instance. By increasing this number you can make sure that the control's id does not clash with any of the controls in the definition

Attributes not set at creation time can be set later. For example, you can test and debug a definition before designing its icon.

You can use the "Prime" button to fill in several of the fields for the currently selected widget.

Base Directory

If a definition is specified with a relative file name (a name that does not start with /), Sun WorkShop Visual adds the *base directory* to the front of the file name. If a base directory is not specified, the directory that contains the definitions design file is used.

To specify a base directory, display the Edit Definitions dialog, click on “Base Directory”, select a new directory and click on “Apply”. The new base directory is saved in your definitions file and is immediately used in the current session of Sun WorkShop Visual. The base directory cannot be changed if the current design contains instances of existing definitions.

Modifying a Definition

Widgets in the definition are frozen. You cannot add or delete widgets, rename them, set constraints on them in the layout editor, or reset resources. To modify a definition, you must temporarily undefine it. When you need to modify a definition, use the following steps:

1. **Open the save file that contains the definition.**
2. **Select the root widget of the definition.**
3. **Pull down the Widget Menu and turn off the “Definition” toggle.**

   Turning off the toggle unfreezes the widgets in the definition so you can make any necessary changes. After making your edits:

4. **Select the root widget and set the “Definition” toggle on again.**
5. **Regenerate the code file and externs file.**
6. **Save the design.**

Impact of Modifying a Definition

Changing a definition affects every design file that uses it. Each time you open a design that uses a definition, Sun WorkShop Visual also opens the file that contains the definition and merges information from the two files. If the definition has been modified, Sun WorkShop Visual tries to reconcile the new definition with the design that uses the old version of it.

If any changes cannot be reconciled, Sun WorkShop Visual displays an error message and saves any irreconcilable parts of the design in temporary Sun WorkShop Visual clipboard files. At this stage there are several ways to proceed:

- Paste the clipboard file into your design and manually resolve its contents with the new definition
- Discard the clipboard file contents altogether

- Exit from Sun WorkShop Visual without saving and modify or revert the definition so that it is compatible with the designs that use it

To minimize the risk of incompatibilities:

- Avoid changing the names of widgets in the definition
- Replace a widget in the definition only with a subclass widget of the same name. For example, replacing a Label “foo” with a PushButton “foo” is normally safe

Instances

To create an instance of a definition simply click on the appropriate button in the palette. The instance is shown with a colored background.

Definitions and instances must be in separate designs. Although you can see them in the same design within Sun WorkShop Visual, the generated code does not compile unless they are separate.

Definition Families

Definitions are grouped together on the widget palette according to their *family*. An option menu above the definitions on the widget palette allows you to change which family is currently displayed. See “Editing the Definitions File” on page 269 for details on specifying a definition’s family.

Modifying and Extending an Instance

Creating an instance of a definition corresponds to creating an instance of the structure (either a C structure or a C++ class). You can modify an instance after you have created it provided that the modifications can be reflected in the generated code. For example, you can set resources on widgets or add children to widgets only if they are accessible (i.e. if they are named and, for C++, they have an appropriate access mode). You cannot remove widgets or change their names. The root widget is an exception. Because the root widget of the instance is always accessible (through the member function *xd_rootwidget()*), it can always be modified.

Note – You cannot move a widget in the layout editor, or specify constraints for it, unless it is accessible.

Creating a Derived Structure

It is frequently useful to create a new structure that is derived from the definition. To do this simply set the Structure option on the Code generation page of the Core resources dialog. The derived structure can only be set to the same value as the definition, e.g. it is not possible to derive a C++ class from a C structure.

Overriding a Definition Callback Method

Inherited methods from definitions can be overridden in the instance so that the instance has different behavior from that specified in the definition.

Compiling Code Containing an Instance

To compile code generated from a design containing an instance of a definition, you need to link in the definition code too. There are two ways to do this:

1. Link in a library containing the definition code
2. Compile the definition code and the instance code together

These are explained separately below.

Using a Library

To link a library containing the definition code in with the instance code, first compile the code for the definition into a library. Usually, on UNIX and using C or C++, this is done in the following way:

```
make <definitioncode>.o
ar r <definitionlib>.a <definitioncode>.o
```

You then need to edit your Makefile for the code containing the instance so that:

1. The compiler can locate the header file for the definition. Sun WorkShop Visual automatically #includes this header file into the code generated for the instance.
2. The linker can locate the library containing the definition code. Simply add the full pathname of the library to “EXTRALIBS”.

Compiling the Definition with the Instance

Another way of compiling the instance of a definition involves generating the definition, the instance of it and a corresponding Makefile into the same directory. You can tell Sun WorkShop Visual to configure the Makefile so that the definition and instance can both be compiled into the same application. The following instructions show you how to do this.

1. **Open the design containing the instance first.**
2. **Make sure you are generating a "Main program"**
3. **Set the "New" and "Template" toggles in the Makefile Options dialog.**
4. **Generate all the required files.**
5. **Open the definition design.**
6. **Unset the "Main Program" generate toggle.**
7. **Unset the "New" toggle in the Makefile options dialog, leaving the "Template" toggle on.**
8. **In the Code Options dialog, set "Links" to None (you have already generated the links functions, doing so twice would result in a linker error).**
9. **Generate the code, externs and Makefile (and resources if required).**
10. **Type**:

    ```
    make
    ```

    at the command prompt.

    Doing the above will give you one application containing your instance.

Definitions and Resource Files

Resource values for widgets that are components of definitions can be either hard-coded or specified in resource files.

Instances and Definition Resource Files

When you specify a resource file for a definition, Sun WorkShop Visual *#includes* that file in the resource file for any design that contains an instance of the definition. The Xlib mechanisms that read the resource file interpret this directive and use it to find the resource file for the definition.

Online Help for Definitions

To record information about a definition and communicate with other developers who are using it, you can provide online help for definitions. The online help is accessed in the Sun WorkShop Visual interface by using the <*Tab*> and arrow keys to get to the icon or button for the definition, then pressing the <*osfHelp*> key (usually <*F1*>).

Help files are stored in subdirectories of the Sun WorkShop Visual help directory. The help directory is determined by the *helpDir* resource. By default, it is

`$VISUROOT/lib/locale/${LANG}/help`

where VISUROOT *is the path to the* Sun WorkShop Visual *installation root directory and* `LANG` *is the name of your locale (default* `C`*).*

Text Help Documents

Text help documents are in HTML format. The name of the file is formed by concatenating the document name and marker name. These are joined using the value of the visu.*userHelpCatString* resource. By default this resource is set to "." The file is then given a ".html" suffix. Sun WorkShop Visual looks for this file in the *UserDocs* subdirectory of the Sun WorkShop Visual help directory.

CHAPTER 9

C++ Code Tutorial

Introduction

This chapter describes how to use Sun WorkShop Visual’s C++ code generation facilities to add structure to application code and to create reusable widget hierarchies that correspond to C++ classes. These reusable hierarchies, known as *definitions*, appear on the widget palette and can be added to the hierarchy like any other widget. Although this chapter primarily covers C++, most of the material covered is also relevant to structured code generation in C with the exception of the sections on callback methods. Where they diverge the differences are noted.

This chapter is a tutorial. It contains step-by-step instructions that show you how to:

- Create a C++ class corresponding to a widget hierarchy
- Use class methods to handle callbacks
- Use derived classes and preludes to add extra members to the generated class
- Modify or replace the base classes from which the Sun WorkShop Visual classes are derived
- Turn a class into a reusable definition and place the definition on the widget palette
- Modify the definition
- Create and modify an instance of the definition
- Use a derived class to extend an instance of the definition
- Override callback methods
- Generate and use resource files for definitions

For best results, read this chapter at your computer while running Sun WorkShop Visual and do the steps as you read.

Further information on the subject of structured code generation can be found in Chapter 8 ”Structured Code Generation and Reusable Definitions“.

Creating a C++ Class

A C++ class in Sun WorkShop Visual corresponds to any widget with its children. When you designate a widget as a C++ class, Sun WorkShop Visual generates a class with that widget and its named descendant widgets as data members. This class can be extended by adding data members and member functions and thus provides a single location for properties that relate to the whole hierarchy.

Designating a C++ Class

Use the following steps to create a widget hierarchy containing a MenuBar widget and designate the MenuBar as a C++ class. Note that this example would *not* be compatible with Microsoft Windows code generation.

1. **Create two new directories,** *libmenu* **and** *cmd* **which have the same parent directory.**
2. **Change to the** *libmenu* **directory and start Sun WorkShop Visual.**
3. **Build the widget hierarchy shown in Figure 9-1.**

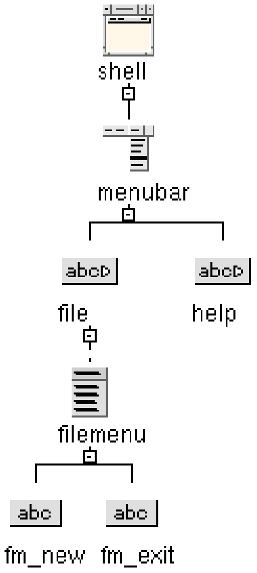


FIGURE 9-1 MenuBar Widget Hierarchy

Only explicitly named widgets are created as members of the class; unnamed widgets are local to the function that creates them. Naming the widgets makes them directly accessible from member functions of the class. Since they are protected members by default, they are also accessible from member functions of any derived class.

4. **Assign the widget names as shown in Figure 9-1.**

5. **Set the "Label" resource for each CascadeButton and PushButton to an appropriate string: "File", "Help", "New" and "Exit".**

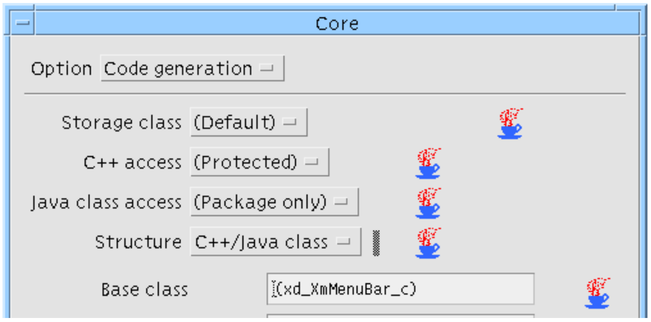
6. **Use the Shell resource panel to designate the Shell widget as an Application Shell. Assign the title "Demo" to the Shell widget.**

   This completes the example hierarchy. Now designate the menu bar as a C++ class:

7. **Select the MenuBar widget in the widget hierarchy.**

8. **Display the "Code generation" page of the Core resource panel.**

9. **Select "C++/Java class" from the "Structure" option menu, as shown in Figure 9-2.**



Core

Option Code generation

Storage class (Default)

C++ access (Protected)

Java class access (Package only)

Structure C++/Java class

Base class (xd_XmMenuBar_c)

FIGURE 9-2 Designating a Widget as a Class

10. **Click on “Apply”.**

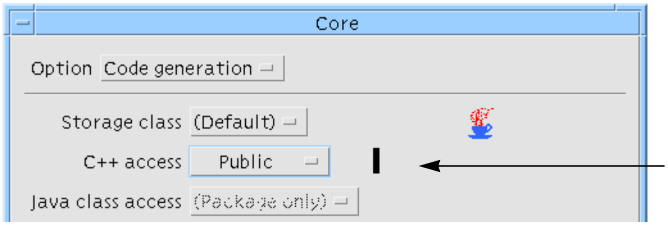
    By default, Sun WorkShop Visual uses the variable name of the widget as the basis for a default class name and “Instantiate as” name and so the widget named *menubar* produces the class named *menubar_c*. The base class for *menubar_c* depends on the widget class; here it is *xd_XmMenuBar_c*. These names can be changed, as described later in this chapter.

Widget Member Access Control

The generated *menubar_c* class will contain the class widget (*menubar*) and all its named descendent widgets as members. Although, by default, they are protected members, you can change the access control on any widget by using the Core resource panel. Use the following steps to make the Help menu a public member of the class.

1. **Select the CascadeButton named *help* in the widget hierarchy.**
2. **Display the “Code generation” page of the Core resource panel.**

   This is shown in Figure 9-3.



Core

Option Code generation

Storage class (Default)

C++ access Public

Java class access (Package only)

“C++ Access” option menu

FIGURE 9-3 Member Access Control

3. **Select “Public” from the “C++ Access” option menu, then click on “Apply”.**

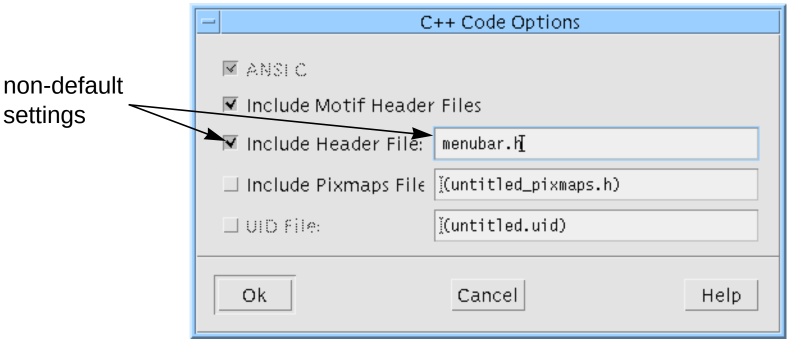
C++ Class Code Generation

The code generated for the class consists of a class declaration in the generated Externs file and an implementation in the primary C++ code file. To generate these files for the example:

1. **Display the Generate dialog and make sure the “Language” option menu is set to C++.**
2. **Type** *menubar.cpp* **into the text box labelled “Code” and set the “Generate” toggle.**
3. **Type** *menubar.h* **into the text box labelled “Externs” and set the “Generate” toggle.**
4. **Type** *menubar.cpp* **into the text box labelled “Main Program” and set the “Generate” toggle.**
5. **Type** *Makefile into the “Makefile” field and set the “Generate” toggle***.**
6. **Click on the “Options” button next to the “Makefile” field.**

   This displays the Makefile options dialog.
7. **In the Makefile Options dialog set both “New makefile” and “Makefile template” toggles on. Press the “Ok” button.**
8. **Press the “Options” button next to the “Code” field.**

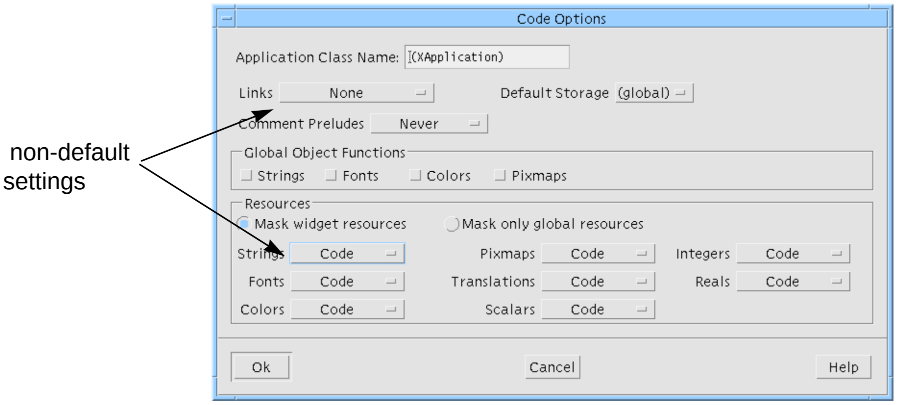
   This produces the Code Options dialog, as shown in Figure 9-4.



☑ ANSI C
☑ Include Motif Header Files
☑ Include Header File: menubar.h
□ Include Pixmaps File (untitled_pixmaps.h)
□ UID File: (untitled.uid)

FIGURE 9-4 Code Options Dialog

9. **Type** *menubar.h* **into the text box labelled “Include Header File”, as shown in Figure 9-4 and set the toggle. Press the “Ok” button.**
10. **Press the “Options” button at the bottom of the Generate dialog and set the options as shown in Figure 9-5. Ensure that the String resources are generated into the Code. Press the “Ok” button.**



Application Class Name: (XApplication)

Links: None Default Storage: (global)

Comment Preludes: Never

Global Object Functions: □ Strings □ Fonts □ Colors □ Pixmaps

Resources: ● Mask widget resources ○ Mask only global resources

Strings	Code	Pixmaps	Code	Integers	Code
Fonts	Code	Translations	Code	Reals	Code
Colors	Code	Scalars	Code		

FIGURE 9-5 Code Generation Options for the *menubar_c* class

11. Press “Generate” in the Generate dialog.

The C++ externs file, *menubar.h*, contains the declaration for the class:

```
...
class menubar_c: public xd_XmMenuBar_c {
public:
    virtual void create (Widget parent, char *widget_name =
         NULL);
    Widget help;
protected:
    Widget menubar;
    Widget file;
    Widget filemenu;
    Widget fm_new;
    Widget fm_exit;
};

typedef menubar_c *menubar_p;
...
```

The new class for this MenuBar is based on an existing class, *xd_XmMenuBar_c*. The MenuBar and its named widget descendants are protected members, except for the widget *help*, which you designated as public.

The primary C++ file, *menubar.cpp*, contains the creation function for the new class. This function creates the MenuBar widget and its descendants. Note that this is *not* done in the constructor for *menubar_c*. This gives you the option of creating the widgets later than the class instantiation.

```
...
#include “menubar.h”
...
void menubar_c::create (Widget parent, char *widget_name)
{
    Widget children[2];/* Children to manage */
    Arg al[64];/* Arg List */
    register int ac = 0;/* Arg Count */
    XmString xmstrings[16];/* temporary storage for
                                    XmStrings */
    if ( !widget_name )
          widget_name = "menubar";
    menubar = XmCreateMenuBar ( parent, widget_name, al,
          ac);
    _xd_rootwidget = menubar;
    xmstrings[0] = XmStringCreateLtoR("File",
          (XmStringCharSet)XmFONTLIST_DEFAULT_TAG);
    XtSetArg(al[ac], XmNlabelString, xmstrings[0]); ac++;
    file = XmCreateCascadeButton ( menubar, "file", al,
          ac);
    ac = 0;
...
    children[ac++] = file;
    children[ac++] = help;
    XtManageChildren(children, ac);
    ac = 0;
}
```

The *menubar.cpp* file also includes a creation function for the complete hierarchy. This function creates any widgets not in the class: in this case, just the Shell. It then creates an instance of the *menubar_c* class and calls *menubar_c::create()* to create the widget members of the class:

```
void create_shell (Display *display, char *app_name, int app_argc,
char **app_argv)
```

```
{
    Widget children[1]; /* Children to manage */
    Arg al[64]; /* Arg List */
    register int ac = 0; /* Arg Count */
    XtSetArg(al[ac], XmNallowShellResize, TRUE); ac++;
    XtSetArg(al[ac], XmNtitle, "Demo"); ac++;
    XtSetArg(al[ac], XmNargc, app_argc); ac++;
    XtSetArg(al[ac], XmNargv, app_argv); ac++;
    shell = XtAppCreateShell ( app_name, "XApplication",
         applicationShellWidgetClass, display, al, ac );
    ac = 0;
    menubar = new menubar_c;
    menubar->create ( shell, "menubar" );
    XtManageChild ( menubar->xd_rootwidget());
}
```

Compiling the Generated C++ Code

Since you set the “Main program” toggle when you generated code, *menubar.cpp* also contains a main program and so the application can be built as it stands.

The C++ code generated by Sun WorkShop Visual is straightforward to build. The only special feature is that the base classes from which the generated classes are derived, such as *xd_XmMenuBar_c*, must be available. The *$*VISUROOT*/src/xdclass/lib* directory contains source for the default base classes. *$*VISUROOT*/src/xdclass/h* contains header files.

If the *libxdclass.a* library has not yet been built, use the following steps to build it using the supplied Makefile:

1. **Go to the *src/xdclass/lib* directory in your Sun WorkShop Visual installation.**
2. **Set VISUROOT to the path of the root of your Sun WorkShop Visual installation.**
3. **Type: `make`**

   When this completes, the *libxdclass.a* library is ready to use.

   You are now ready to build the program using the generated Makefile.

Note – Because the generated Makefile contains references to $VISUROOT, you must set this environment variable before building the program.

---

4. **To build the menubar program, type:** `make`

5. **To run the application, type:** `menubar`

   The application looks and behaves exactly as it would if there were no classes in it.

Callback Methods

So far, this example has shown how to designate a widget hierarchy as a class and the form of the code that is generated. At this stage, it does not exploit the fact that the widget hierarchy is a class.

Note – This section, and the following three sections, are specific to C++ programming. When using C structures the conventional callback mechanism applies. If you are not doing C++ programming you might like to skip to "Creating a Definition" on page 297.

Callbacks and Member Functions

Sun WorkShop Visual provides a simple mechanism that allows you to specify class member functions as callback functions. In Sun WorkShop Visual these are known as callback *methods*. The technique used is discussed fully in "Callback Methods" on page 259.

Specifying a Callback Method

The callbacks dialog lets you specify the member functions that are invoked in response to events. When you specify a callback method for a particular widget, the method which is invoked is that which belongs to the most immediate class-designated ancestor of the widget (perhaps the widget itself). For example, callback methods on the menu buttons in the MenuBar example invoke member functions of the *menubar_c* class.

Use the following steps to declare a class method on the *fm_new* button:

1. **Select the** *fm_new* **button.**

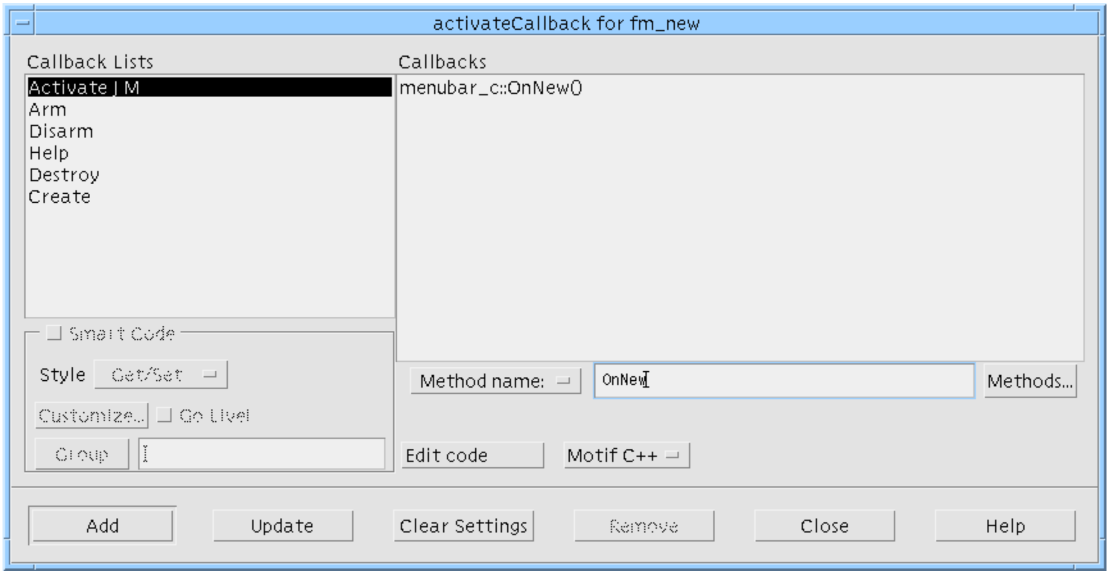
2. **Invoke the Callbacks dialog by selecting "Callbacks" from the "Widget" menu or pressing the Callbacks button on the toolbar.**

3. **Select “Activate” from the list of callback lists.**

4. **In the “Method name:” field, type `OnNew`**

5. **Click on “Add”.**

   This adds *OnNew* to the list of local methods, as shown in Figure 9-6:



activateCallback for fm_new

Callbacks: menubar_c::OnNew()

Method name: OnNew

FIGURE 9-6 Specifying a Callback Method

This designates the member function *menubar_c::OnNew()* as the method that handles the Activate callback of the button *fm_new*. When you do this, Sun WorkShop Visual also declares the method on the parent widget *menubar* if you haven’t already declared it.

Use a similar procedure to enter a callback method on the *fm_exit* button:

6. **Select the *fm_exit* button in the widget hierarchy.**

7. **Select “Activate” from the Callbacks dialog.**

8. **In the “Method name:” field, type: `OnExit`**

9. **Click on “Add”.**

10. **Close the Callbacks dialog**

    The class now has two callback methods, *menubar_c::OnNew()* and *menubar_c::OnExit()*.

Generating Code for Callback Methods

Using a callback method from within a class causes Sun WorkShop Visual to generate declarations for two additional member functions, a complete implementation for one of them and a stub for the other.

1. **Display the Generate dialog and make sure the "Language" option menu is set to C++.**

2. **Type** *menubarS.cpp* **into the text box labelled "Stubs" and set the "Generate" toggle.**

3. **Display the Makefile Options dialog.**

4. **Set the "New makefile" toggle off and the "Makefile template" toggle on. Press the "Ok" button.**

5. **Set the Makefile "Generate" toggle in the Generate dialog.**

6. **Press the "Generate" button.**

Look at the class declaration in *menubar.h*. Two new member functions have been added for each callback method:

```
class menubar_c: public xd_XmMenuBar_c {
public:
...
    static void OnExit( Widget, XtPointer, XtPointer );
    virtual void OnExit( Widget, XtPointer );
    static void OnNew( Widget, XtPointer, XtPointer );
    virtual void OnNew( Widget, XtPointer );
};
```

Note that only the static versions of these functions have the argument list expected by an Xt callback. Therefore, when Xt invokes the callback method *menubar_c::OnNew()*, the C++ compiler selects the static version based on the argument list.

Note the following line in the creation function in *menubar.cpp*:

```
XtAddCallback (fm_new, XmNactivateCallback, OnNew, (XtPointer)
    this);
```

The code for the static function is also generated into *menubar.cpp*. This function simply invokes the non-static virtual member *OnNew(Widget, XtPointer)*, using the instance pointer passed in as client data:

```
void menubar_c::OnNew( Widget widget, XtPointer client_data,
XtPointer call_data )
```

```
{
    menubar_p instance = (menubar_p) client_data;
    instance->OnNew ( widget, call_data );
}
```

You provide the code for the non-static virtual member function *OnNew(Widget, XtPointer).* A stub for this function is generated to *menubarS.cpp*:

```
void
menubar_c::OnNew (Widget w, XtPointer xt_call_data )
{
    XmAnyCallbackStruct *call_data = (XmAnyCallbackStruct*)
          xt_call_data;
}
```

Sun WorkShop Visual generates code according to this pattern for all the callback methods that are used in a hierarchy. In this example, similar code is generated for *OnExit()*.

OnNew() and *OnExit()* are invoked from the *fm_new* and *fm_exit* PushButtons but the functions are methods of the *menubar_c* class. This means that all the callback functions that define the behavior of widgets in the class are kept in one place. It also means that all callback functions have access to the instance data for the class and can use it to share information.

Implementing a Callback Method

The application behavior is added by implementing the callback methods. You can edit the callback method using Sun WorkShop Visual's editing mechanism. Use the following steps to implement the *OnExit()* method:

1. **Select the *fm_exit* button.**

2. **Display the Callbacks dialog and select the *OnExit()* callback method.**

3. **Press the "Edit code" button.**

   The file *menubarS.cpp* is opened ready for you to add your code to the OnExit method. See "Editing Callback Code from Within Sun WorkShop Visual" on page 226 for more details on callback editing.

4. **Complete the implementation of *menubar_c::OnExit()* as:**

   ```
   void
   menubar_c::OnExit (Widget w, XtPointer xt_call_data )
   {
   ```

```
        XmAnyCallbackStruct *call_data = (XmAnyCallbackStruct*)
              xt_call_data;
        exit(0);
   }
```

5. **Close the Callbacks dialog.**
6. **To build the menubar program, type: `make`**
7. **To run the application, type: `menubar`**
8. **Select the "Exit" button from the File Menu.**

   Verify that the program exits.

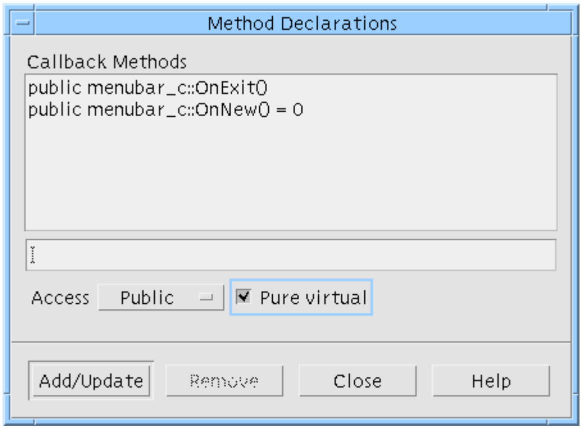
Editing Methods Attributes

Callback methods have two attributes: their access level and whether they are designated pure virtual. The access level determines whether the method is accessible from derived classes and external code. A method can be designated pure virtual to indicate that it has no implementation in the base class, which must be sub-classed to provide an implementation. See "C++ Classes" on page 254 for further details.

The attributes are set initially to default values when the callback method is first specified. This can be done on any widget that has a class ancestor or is a class itself. However, they can only be changed on the root widget for the class. For example, the *OnNew()* callback method of the class *menubar_c* can only be edited via the *menubar* widget itself.

1. **Select the *menubar* widget in the widget hierarchy.**
2. **Select "Method declarations" from the "Widget" menu to display the method declarations dialog.**

   This shows a list of the callback methods declared for the class (rather than a list of the methods invoked for any particular event). You can use this panel to edit the callback methods and to declare methods that are not invoked by events in this class but which may be invoked in a derived class.
3. **Select the method *OnNew()* and set the "Pure virtual" toggle.**
4. **Click on the "Add/Update" button to apply the change. The result is shown in Figure 9-7.**



Callback Methods

public menubar_c::OnExit()
public menubar_c::OnNew() = 0

Access Public ☑ Pure virtual

FIGURE 9-7 Editing a Callback Method

5. **Generate the code again.**

   You do not have to provide an implementation for a pure virtual member function although it is legal to do so. Therefore:

6. **Copy the stubs file,** *menubarS.cpp***, to** *temp.cpp***.**

7. **Edit the stubs file,** *menubarS.cpp***, to remove the stub for** *OnNew()***, i.e. the code between the curly braces.**

8. **Build the menubar program, as before.**

   The C++ compiler produces an error message like:

   ```
   "menubar.cpp" line 100: Error: Cannot create a variable for abstract
   class menubar_c
   ```

   The error occurs because the *menubar_c* class contains a pure virtual function and therefore cannot be instantiated. It is now only useful as a base class. Later in this chapter you will use this class as a basis for a derived class.

9. **Copy the file** *temp.cpp* **to** *menubarS.cpp* **and then remove** *temp.cpp***.**

Adding Class Members

This section and the following one present two techniques for adding members to Sun WorkShop Visual's generated classes. The two techniques are:

- Using Sun WorkShop Visual's preludes mechanism
- Generating a derived class

Note – This section is specific to C++ programming.

Adding Class Members as a Prelude

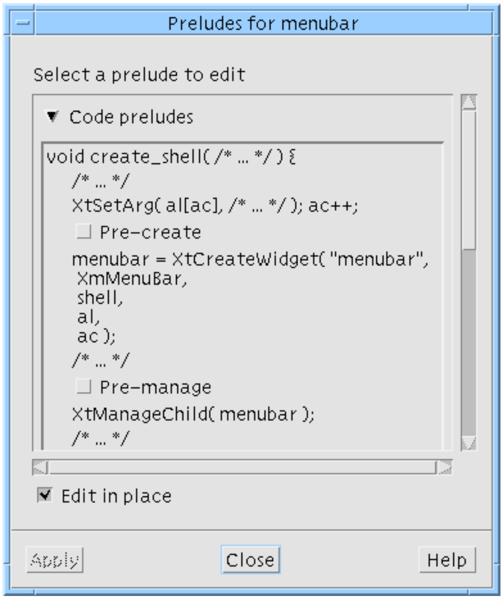
The easiest way to add a small number of members is to use the preludes mechanism. This lets you type fragments of code in Sun WorkShop Visual and have them passed into the generated code.

Note – It is possible to type preludes directly into the generated code using Sun WorkShop Visual's edit mechanism. See "Customizing the Generated Files: Preludes" on page 239 for details on doing this.

For this example, we shall add the prelude in the Preludes dialog before generating the code.

1. **Select the *menubar* widget in the widget hierarchy.**
2. **Pull down the Widget Menu and select "Code preludes".**

   This displays the dialog shown in Figure 9-8.



```
void create_shell( /* ... */ ) {
    /* ... */
    XtSetArg( al[ac], /* ... */ ); ac++;
     □ Pre-create
    menubar = XtCreateWidget( "menubar",
     XmMenuBar,
     shell,
     al,
     ac );
    /* ... */
     □ Pre-manage
    XtManageChild( menubar );
    /* ... */
```

FIGURE 9-8 Code Preludes Dialog

3. **Unset the “Edit in place” toggle.**

   Doing this expands the Preludes dialog so that an editing area appears on the right. This is where you will add the prelude.

4. **Select the “Protected methods” toggle in the text labelled “Method Preludes”. (You may have to scroll down to find this.)**

5. **In the text area on the right, press the TAB key and then type**:

   ```
       int modified;
   ```

   and press Return.

6. **Click on “Apply” and then “Close”.**

   To see the result of this operation:

7. **Generate the code again.**

8. **Examine** *menubar.h* **and verify that the class** *menubar_c* **now has the additional member.**

Creating a Derived Class

The Code Preludes dialog is designed for making small insertions to the generated code. To add substantial functionality, it is often better to write a new class derived from the generated class. The logical gap between the two classes can be used to add members and provide implementations for virtual functions.

Note – This section is specific to C++ programming.

By default, Sun WorkShop Visual derives the name of a C++ class from the variable name of the root widget and so the class for the widget *menubar* is *menubar_c*:

```
class menubar_c: public xd_XmMenuBar_c {
...
};
```

When Sun WorkShop Visual generates code to create an instance of the class, it uses the same name:

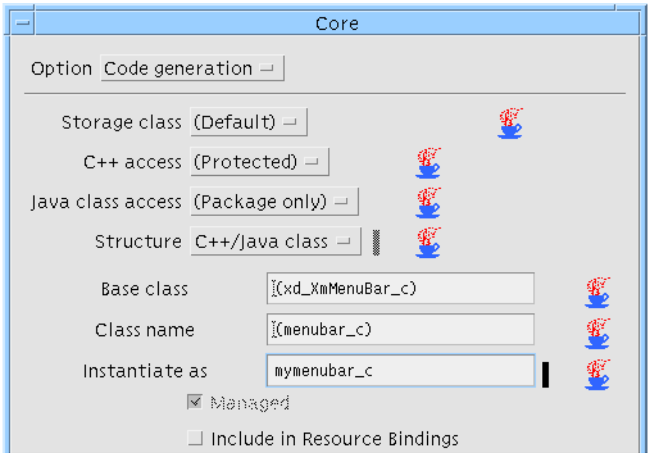
```
menubar = new menubar_c;
```

You can change the default behavior so that Sun WorkShop Visual declares the generated class under one name and creates the instance under another, for example:

```
menubar = new mymenubar_c;
```

To make this change, use the “Instantiate as” field on the Code Generation page of the Core resource panel:

1. **Select the *menubar* widget in the widget hierarchy.**
2. **Display the “Code generation” page of the Core resource panel.**
3. **Set the “Instantiate as” name to *mymenubar_c,* as shown in Figure 9-9.**



Option Code generation

Storage class (Default)

C++ access (Protected)

Java class access (Package only)

Structure C++/Java class

Base class (xd_XmMenuBar_c)

Class name (menubar_c)

Instantiate as mymenubar_c

☑ Managed

□ Include in Resource Bindings

FIGURE 9-9 Changing "Instantiate as" Name

4. **Click on "Apply" and then "Close".**

5. **Generate the code again.**

The original class, *menubar_c*, is declared exactly as before. However, when Sun WorkShop Visual generates code to create an instance of the class, it uses the "Instantiate as" name:

```
menubar = new mymenubar_c;
```

Writing the Derived Class

Sun WorkShop Visual doesn't generate code for the *mymenubar_c* class. You must provide a header file which declares the class and code to implement any methods it contains. There are no limitations on the new class except that it must be derived from *menubar_c*. For this example use the sample code given below.

1. **In a new file named** *mymenubar.h*, **write the class declaration for the derived class** *mymenubar_c*.

Use the following code:

```
#ifndef _mymenubar_h
#define _mymenubar_h
#include <menubar.h>
class mymenubar_c: public menubar_c {
public:
// Constructor
```

```
    mymenubar_c();
     //Provide implementation for inherited pure virtual
    void OnNew(Widget, XtPointer);
};
#endif
```

Because the new class is derived from *menubar_c*, it inherits all widget members and member functions you declared for that class in Sun WorkShop Visual. You can add any number of new members. Here we add a constructor function and an implementation of the *OnNew()* virtual callback method.

2. **In a new file named** *mymenubar.cpp* **write the class implementation for the derived class** *mymenubar_c***.**

   Use the following code:

```
#include <mymenubar.h>
mymenubar_c::mymenubar_c()
{
    modified = TRUE;
}
void
mymenubar_c::OnNew(Widget, XtPointer)
{
     // Reset modified flag
    if (modified)
         modified = FALSE;
}
```

This completes all the code for the class. Note that the generated C++ code module *mymenubar.cpp* needs to include the header file for the derived class *mymenubar_c*. This is done using the “Include Header File” in the Generate dialog.

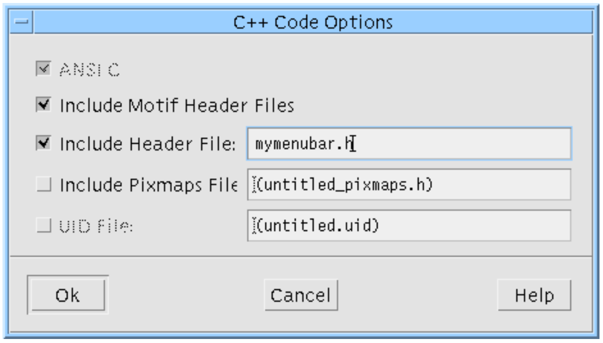
3. **Display the Generate dialog.**

4. **Press the “Options” button next to the “Code” field.**

   This displays the Code Options dialog.

5. **Set the “Include Header File” field to** *mymenubar.h***, set the associated toggle and click on “Ok”.**

   The Code Options dialog containing the new file name is shown in Figure 9-10:



☑ ANSI C

☑ Include Motif Header Files

☑ Include Header File: mymenubar.h

□ Include Pixmaps File (untitled_pixmaps.h)

□ UID File: (untitled.uid)

FIGURE 9-10 Changing the Declarations Header

6. **Generate the code.**
7. **Add the following lines to the Makefile before the line that reads “XD_ALL_C_SOURCES=...”.**

```
XD_CC_SOURCES=mymenubar.cpp
XD_CC_OBJECTS=mymenubar.o
```

8. **Add the following lines at the end of the Makefile:**

```
mymenubar.o:mymenubar.cpp
     $(CCC) $(CCFLAGS) $(CPPFLAGS) -c mymenubar.cpp
```

Note – The indentation of the compiler instruction line is deliberate, you must have this too.

---

9. **Save the Makefile.**
10. **Build the menubar program, as before.**

The application now uses the class *mymenubar_c* and invokes its *OnNew()* method when the “New” button in the “File” menu is pressed. To check this, you can extend *mymenubar_c::OnNew()* to print out a message.

You should note that actual parameters for the constructor can be supplied with the class name. For example, setting the “Instantiate as” string to mymenubar_c (“Hello World”) will cause Sun WorkShop Visual to generate:

```
menubar = new mymenubar_c ( "Hello World" );
```

Creating a Definition

Once a hierarchy of widgets has been encapsulated as a class, you can re-use it in other designs by turning it into a *definition*. A definition is a reusable group of widgets which can be added to the Sun WorkShop Visual widget palette. Selecting a definition from the palette creates an instance of that structure in the design. When Sun WorkShop Visual generates code containing an instance, the definition's create function is called to create the widgets.

Prerequisites

A hierarchy of widgets can become a definition provided that:

- The root widget has a variable name
- The root widget has been designated as a C++ class or a C Data Structure
- The root widget is not part of another definition
- The widget hierarchy does not contain a definition
- The widget hierarchy does not contain any global or static widgets

Designating a Definition

Use the following steps to designate the MenuBar class as a definition:

1. **Select the *menubar* widget in the hierarchy.**

2. **Pull down the Widget Menu and set the "Definition" toggle.**

   The *menubar* widget and its descendants are enclosed in a colored box to indicate that they constitute a definition.

3. **Save the design as *menubar.xd*.**

Note – You must save the design containing the definition before adding it to the palette. Sun WorkShop Visual uses the saved design file each time the definition is used. Although you can have multiple definitions in a single design file, it is easier to keep track if each file contains only one definition.

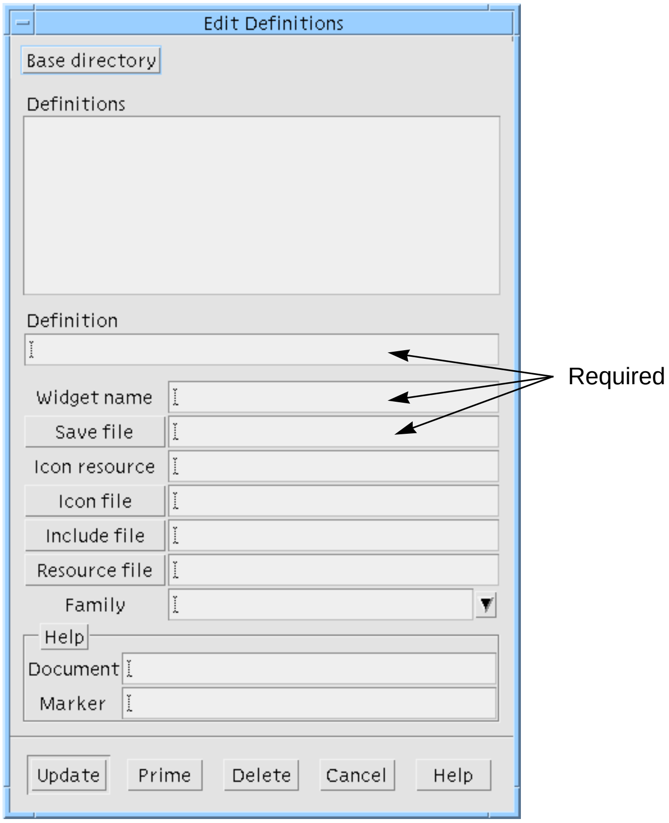
Creating a definition freezes the widgets within it. Their resource panels are disabled and you cannot add widgets or change widget names. You can edit the widgets that make up a definition only by temporarily removing the definition. This should be done with caution to avoid conflicts with designs that use the definition. For details, see “Modifying a Definition” on page 272.

Adding a Definition to the Palette

This section explains how to add the new definition to the widget palette.

1. **Select the menubar widget in the design hierarchy.**
2. **Choose “Edit definitions” from the Palette Menu.**

   This displays the dialog shown in Figure 9-11.



Required

FIGURE 9-11 Adding a Definition to the Palette

You can use this dialog to add a definition (if it has been "marked" as one), delete a definition or edit an existing definition. To add a definition, you must supply:

- *Definition* – A definition name
- *Widget name* – The name of the root widget of the definition
- *Save file* – The name of a saved design file (*.xd*)

Other attributes are described in "Designating a Definition" on page 268.

Attributes not set at creation time can be set later. For example, you can test and debug a definition before designing its icon.

3. **Press the "Prime" button.**

   This fills in the "Save file", "Definition", "Widget Name" and "Include file" fields.

   Note that the "Instantiate as" name you specified for the MenuBar applies each time the definition is used.

4. **Enter** *menubar.xpm* **in the "Icon file" field.**

   The icon is optional. If you do not specify an icon, Sun WorkShop Visual uses the name of the root widget as a label in a PushButton on the widget palette. If you specify an icon file and the icon file does not exist, the Sun WorkShop Visual icon of the widget at the root of the definition is displayed on the palette,

   You can use the Sun WorkShop Visual pixmap editor to design an icon for your definition. It should use an area of color "none" which is used to show the selection in the widget hierarchy. See "Using the Pixmap Editor" on page 161 for details on the pixmap editor and "Palette Icons" on page 703 for details on creating new widget palette icons.

   Note that you can also specify the icon via the Sun WorkShop Visual resource file. To do this, specify the name of the Sun WorkShop Visual resource in the "Icon resource" field and set that resource to a file name in the Sun WorkShop Visual resource file.

   The other fields in the Edit Definition dialog are discussed later in this chapter.

5. **Click on "Update".**

   The icon you specified appears on the Sun WorkShop Visual widget palette. It becomes active whenever you select a widget that can have a MenuBar child.

Generating Code for a Definition

The code for a definition has two parts: the declaration in the public header file (the externs file) and the code module containing the implementation. The code module does not have to be public in order to compile applications containing instances; it can be made available in compiled form in a library.

Use the steps in this section to generate only the code for the definition, i.e. without the Shell or other widgets and without a main program.

To mark the Shell widget so that no code is generated for it:

1. **Select the** *shell* **widget in the hierarchy.**

2. **Display the Core resource panel and select the "Code generation" page.**

3. **Set the "Structure" option menu to "Children only" and then click on "Apply" followed by "Close".**

   After you do this, Sun WorkShop Visual ignores the Shell and any of the Shell's children that are not designated as C++ classes, functions, or data structures. Code is generated only for the MenuBar and its descendants.

   You must also suppress generation of the main program:

4. **Open the Generate dialog.**

5. **Unset the "Generate" toggle for "Main program".**

6. **Ensure that the "Generate" toggles for "Code", "Stubs" and "Externs" are on and that** *menubar.h is specified as the Externs file name***.**

7. **Unset the "Generate" toggle for "Makefile".**

8. **Press "Generate".**

9. **Save the design file.**

   This completes the process needed to create a definition and the code for the corresponding class. It can now be used in an application. The normal way to make the implementation available for reuse is as a library:

10. **Use the following commands to create the library:**

```
make menubar.o
make menubarS.o
make mymenubar.o
ar r libmenu.a *.o
```

Creating an Instance

A definition can be used in the same way as a widget on the palette. Clicking on the palette button creates an instance of the definition. Sun WorkShop Visual copies the definition's hierarchy into the tree where it can be modified and extended. In the generated code, Sun WorkShop Visual will include a call to the definition's creation function to create the instance.

Use the following steps to build a new design using the *menubar* definition.

1. **Select "New" from the File menu.**

2. **Click on the following palette icons: Shell, MainWindow and your new menubar definition.**

   Your new definition is appended to the widget palette and has the icon you specified in Step 4 on page 299.

   This produces the widget hierarchy shown in Figure 9-12.

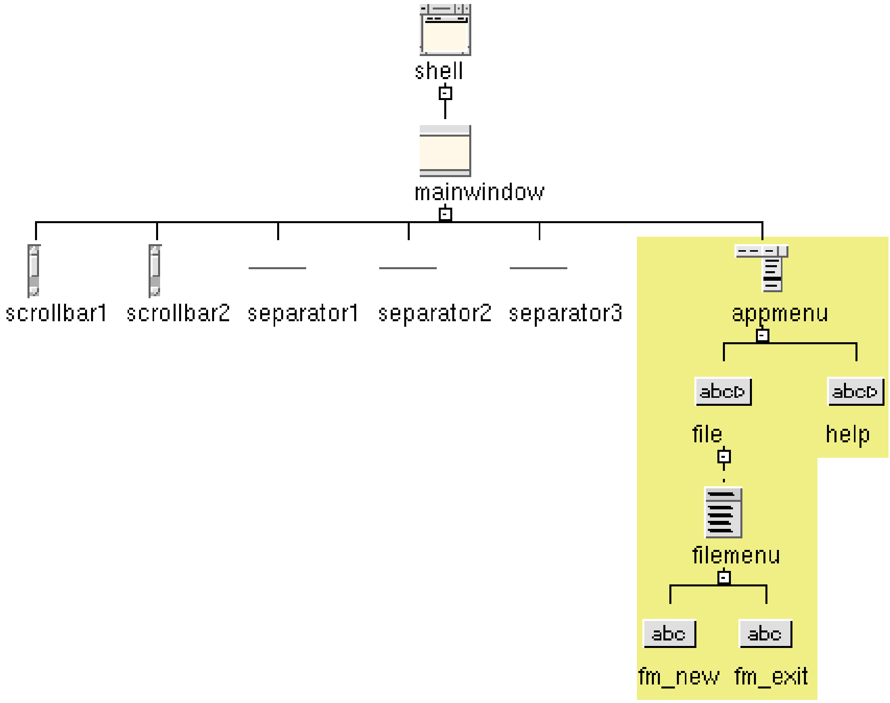


FIGURE 9-12 Hierarchy Containing an Instance of a Definition

The components of the instance are enclosed in a colored box to indicate that they form a single entity. All widgets except the root widget are given the same names they had in the original definition. The root widget is assigned a default name of the form <*widgetclass*><*n*>. For reliable code, assign it an explicit name.

3. **Name the root widget of the instance, the Shell and the MainWindow as shown in Figure 9-12.**
4. **Use the Shell resource panel to designate the Shell widget as an ApplicationShell.**

Modifying and Extending an Instance

You can modify an instance after you have created it provided that the modifications can be done in the generated code. For example, you can set resources on widgets or add children to widgets only if they have public access. You cannot remove widgets or change their names. The root widget is an exception. Because the root widget of the instance can be accessed through the member function *xd_rootwidget()*, it can always be modified.

In our example, all components of the definition are protected except for the *help* button. This means only the *help* button's label can be changed. Similarly, it is possible to add extra widgets under the *help* button but not under the *filemenu* menu.

1. **Select the** *help* **widget in the widget hierarchy.**
2. **Click on the Menu icon in the widget palette and then on the PushButton icon.**

   This adds a single item menu under the *help* CascadeButton.
3. **Set the widget names as shown in Figure 9-13.**

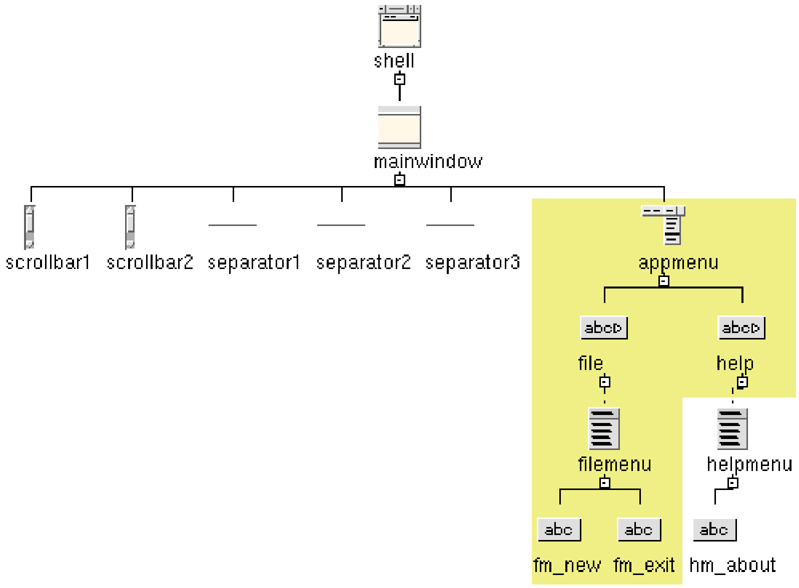


FIGURE 9-13 Extending an Instance of a Definition

4. **Set the label for the** *hm_about* **button to "About...".**

   Currently, you cannot modify other widgets in the definition. For example, you cannot add buttons to *filemenu*. However, if you create a subclass from a definition, you can modify protected as well as public widgets. This technique is discussed in the next section.

Creating a Derived Class

Because most members of the class corresponding to the definition are protected, they cannot be accessed in an instance of the definition. There are two ways to address this:

- modify the original definition to make the members public
- designate the instance as a class (Because the instance class is derived from the definition class, it has access to the protected members)

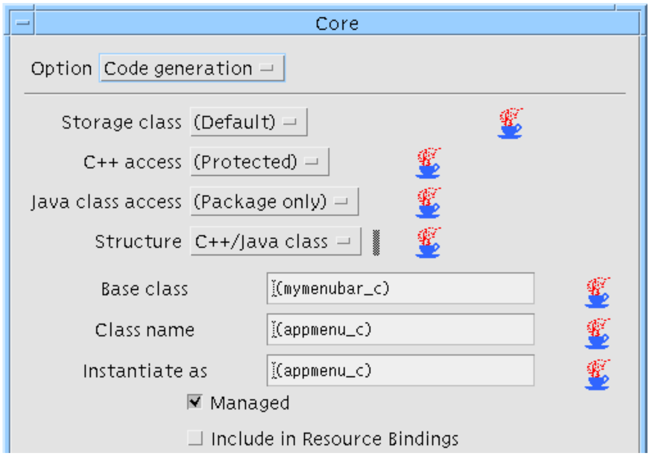
The second approach maintains better encapsulation and lets you exploit the callback methods.

1. **Select the MenuBar widget,** *appmenu*, **in the widget hierarchy.**

2. **Display the “Code generation” page of the Core resource panel.**

3. **Select “C++ Class” from the “Structure” option menu and then click on “Apply”. Close the resource panel.**

   The MenuBar widget is designated as a class, as in Figure 9-14. This class is derived from the class that corresponds to the definition.



Option Code generation

Storage class (Default)

C++ access (Protected)

Java class access (Package only)

Structure C++/Java class

Base class (mymenubar_c)

Class name (appmenu_c)

Instantiate as (appmenu_c)

☑ Managed

□ Include in Resource Bindings

FIGURE 9-14 Creating a Derived Class from a Definition

   Because member functions of the class can access the protected members of *mymenubar_c*, extra widgets can now be added anywhere in the hierarchy.

4. **Select the *filemenu* widget in the widget hierarchy.**

5. **Add two extra PushButtons to the menu and label them “Open...” and “Save...”.**

6. **Set the variable names of the new buttons to *fm_open* and *fm_save*.**

7. **Use mouse button 1 to drag the new buttons into the positions shown in Figure 9-15.**

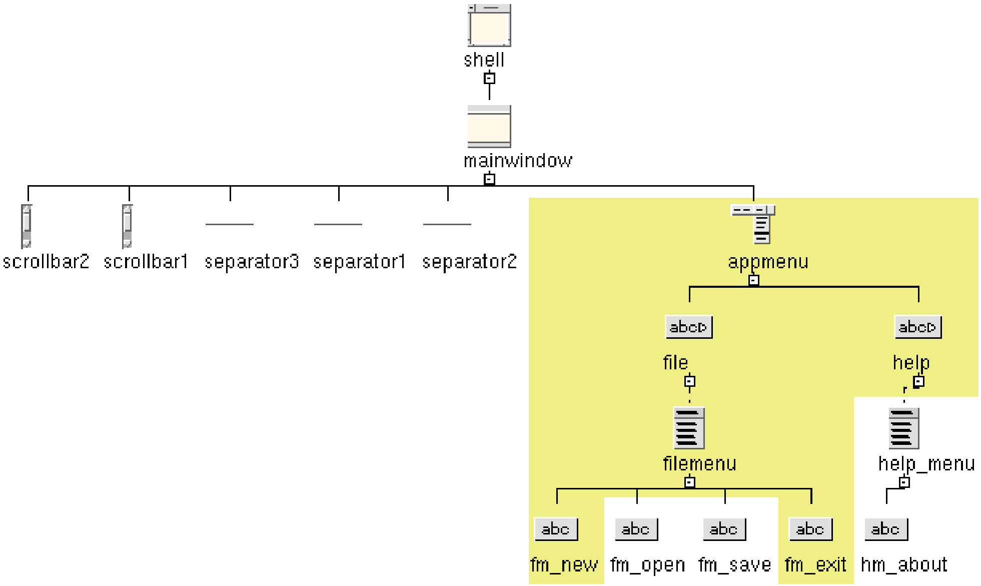


FIGURE 9-15 Extending a Derived Class

The order of a definition cannot be changed. This means that, while you can move new widgets into the definition, you cannot move widgets which are part of the definition.

8. **Select** *fm_open* **in the hierarchy.**
9. **Display the Callbacks dialog either by selecting "Callbacks" from the "Widget" menu or pressing the Callbacks button on the toolbar.**
10. **Select "Activate" from the list of callback lists.**
11. **In the "Method name:" field, type:** `OnOpen`
12. **Click on "Add".**
13. **Repeat the above steps to set the Activate callback methods for** *fm_save* **to** *OnSave*.
14. **Close the Callbacks dialog.**

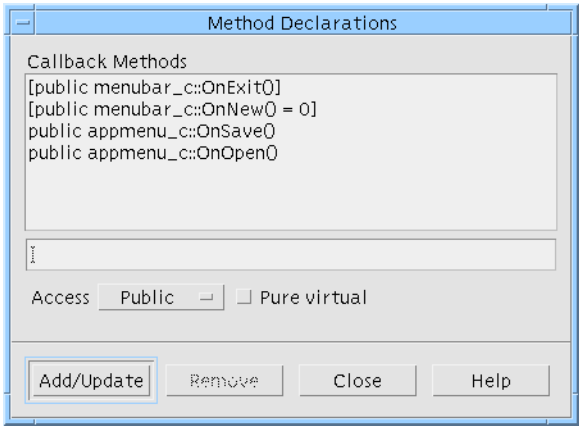
This technique is also valid for C structures. Sun WorkShop Visual will generate a new structure which is an extension of the definition's structure.

Overriding a Callback Method

The class *appmenu_c*, which corresponds to the MenuBar, has four callback methods: *OnNew()* and *OnExit()*, which are inherited, and *OnOpen()* and *OnSave()* which are defined by *appmenu_c* itself. However, the inherited methods can be overridden so that the derived class has different behavior from the base class.

Note – This section is specific to C++ code.

1. **Select the widget *appmenu* in the widget hierarchy.**
2. **Select "Method declarations" from the "Widget" menu.**



```
[public menubar_c::OnExit()]
[public menubar_c::OnNew() = 0]
public appmenu_c::OnSave()
public appmenu_c::OnOpen()
```

Access: Public

□ Pure virtual

FIGURE 9-16 Method Declarations

Note that *OnSave()* and *OnOpen()* are local to *appmenu_c*, whereas *OnExit()* and *OnNew()* are inherited from *menubar_c*. Inherited methods are shown in square brackets.

3. **In the text field, type: `OnExit`**
4. **Make sure the "Pure virtual" toggle button is unset and then press "Add/Update".**

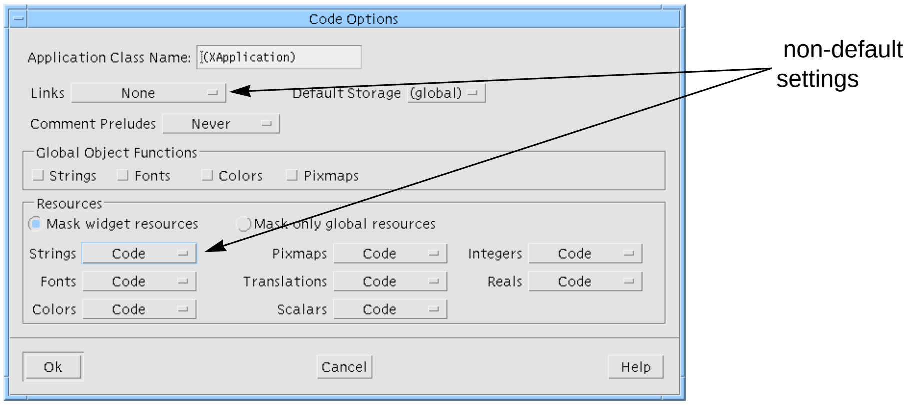
   OnExit() is added to the list of local methods. It can now be overridden.

Implementing Overridden Methods

Overridden methods are implemented by completing the stubs generated by Sun WorkShop Visual:

1. **Display the Generate dialog and make sure the “Language” option menu is set to C++.**
2. **Replace the text in the Directory field by the absolute path of your** *cmd* **directory, e.g.** */u/mgs/TUTORIAL/cmd***.**

   This specifies the name of the directory into which the code will be generated.
3. **Type: `app.cpp` into the text box labelled “Code” and set the “Generate” toggle.**
4. **Type: `appS.cpp` into the text box labelled “Stubs” and set the “Generate” toggle.**
5. **Type: `app.h` into the text box labelled “Externs” and set the “Generate” toggle.**
6. **Type: `app.cpp` into the text box labelled “Main Program” and set the “Generate” toggle.**
7. **Type: `Makefile`** *into the “Makefile” field and set the “Generate” toggle***.**
8. **In the Makefile Options dialog set both “New makefile” and “Makefile template” toggles on and press the “Ok” button.**
9. **Press the “Options” button next to the “Code” field.**
10. **Type: `app.h` into the text box labelled “Include Header File”, set the toggle and press the “Ok” button.**
11. **Press the “Options” button at the bottom of the Generate dialog and set the options as shown in Figure 9-17.**



Code Options

Application Class Name: (XApplication)

Links: None

Default Storage: (global)

Comment Preludes: Never

Global Object Functions

□ Strings □ Fonts □ Colors □ Pixmaps

Resources

● Mask widget resources ○ Mask only global resources

Strings: Code

Fonts: Code

Colors: Code

Pixmaps: Code

Translations: Code

Scalars: Code

Integers: Code

Reals: Code

FIGURE 9-17 Code Generation for the Application

12. **"Ok" the dialog.**

13. **Press "Generate" and close the Generate dialog.**

14. **Complete the stubs file, *appS.cpp*, as follows:**

```
...
#include <iostream.h>
void
appmenu_c::OnExit (Widget w, XtPointer xt_call_data)
{
     ...
     if (modified)
          XBell(XtDisplay(w), 100);
     else
          exit(0);
}
void
appmenu_c::OnSave (Widget, XtPointer xt_call_data)
{
     ...
     cout << “Saving...” << endl;
     modified = FALSE;
}

void
```

```
appmenu_c::OnOpen (Widget, XtPointer xt_call_data)
{
     ...
     cout << “Opening...” << endl;
     modified = TRUE;
}
```

This implementation of *OnExit()* overrides the implementation in the definition. The function *XBell()* rings the bell on the X server. *OnSave()* and *OnOpen()* are stub functions that print appropriate messages and update the *modified* flag.

The functionality of the application’s menubar is now:

- The *modified* flag is initially set *TRUE* in the constructor for the class. It is set *TRUE* by “Open” and set *FALSE* by “New” and “Save”
- “Exit” terminates the application unless the *modified* flag is set, in which case it rings the bell
- “Open” and “Save” produce informative messages on stdout

15. **Edit the Makefile and make the following changes:**

```
MOTIFLIB=-lmenu -lXpm -lXm -lXt -lX11
CFLAGS=-I. ${XINCLUDES} -I${XPMDIR} -I../libmenu \
-L../libmenu
```

16. **Save the Makefile.**
17. **Build the program by typing** `make`
18. **Run the application and verify that the menu behaves as expected.**
19. **Save the design as** *app.xd* **in the** *cmd* **directory.**

Definitions and Resource Files

Resource values for widgets that are components of definitions can be either hard-coded or specified in resource files. When you specify a resource file for a definition, Sun WorkShop Visual includes that file in the resource file for any design containing an instance of the definition.

So far in this example, all resources have been hard-coded. Use the following steps to regenerate the *menubar* definition with string resources in a resource file:

1. **Open the design file** *menubar.xd* **from the** *libmenu* **directory.**
2. **Display the Generate dialog and make sure the “Language” option menu is set to C++ and the Directory is set to the** *libmenu* **directory.**
3. **Display the Code Options dialog from the Generate dialog and set the “Strings” option menu to “Resource file”. Press the “Ok” button.**

   This removes hard-coded string resources such as the labels on buttons.

   Now generate a resource file containing the string resources:
4. **In the Generate dialog, type** *menubar.res into the “X Resources” field and set the corresponding “Generate” toggle.*
5. **Check that the “Generate” toggle for the “Code” field is on and the “Generate” toggles for the “Main program” and “Makefile” are off.**
6. **Press “Generate”.**
7. **Save the design.**
8. **In the libmenu directory, type:** `make clean`

   This removes the old object files.
9. **Repeat Step 10 on page 300 to rebuild the library.**

Editing the Definition

In the preceding steps, you changed the generated code for the definition so that string resources are kept in a resource file. Now edit the *menubar* definition and specify a resource file:

1. **Select “Edit definitions” from the Palette Menu.**
2. **On the Edit Definitions dialog, select** *menubar* **from the scrolled list of definitions.**
3. **In the “Resource file” field, type:**

   ```
   ../libmenu/menubar.res
   ```
4. **Click on “Update”.**

   This step saves the change to the definition in your definitions file (*$HOME/.xddefinitionsrc*).

Note – You can use the Edit Definition dialog to specify a resource file at any time. The original *menubar.xd* design does not have to be loaded to perform this step.

Instances and Definition Resource Files

When you specify a resource file for a definition, Sun WorkShop Visual *#include*s that file in the resource file for any design that contains an instance of the definition. The Xlib mechanisms that read the resource file interpret this directive and use it to find the resource file for the definition. Use the following steps to try this in the *app.xd* application.

1. **Open the design file *app.xd* from the *cmd* directory.**
2. **Display the Generate dialog and make sure the “Language” option menu is set to C++.**
3. **Type: `app.res`** *into the “X Resources” field and set the corresponding “Generate” toggle.*
4. **Set the “Generate” toggles for “Code” and “Main Program”.**
5. **Check that the “Generate” toggles for “Stubs”, “Externs” and “Makefile” are off.**
6. **Display the Code Options dialog from the Generate dialog and set the “Strings” option menu to “Resource file”. Press the “Ok” button.**
7. **Press “Generate”.**

   The X resource file for the application contains the following directive:

   ```
   ! Generated by Sun WorkShop Visual
   #include “../libmenu/menubar.res”
   ```

   Xlib interprets this *#include* directive as giving a pathname relative to the directory containing the application resource file. For details, see the Xlib documentation.

8. **Build the program by typing: `make`**
9. **Set the environment variable XENVIRONMENT to the name of the resource file.**

   The exact syntax for doing this will differ depending on which shell you are using. For a C shell, enter:

   ```
   setenv XENVIRONMENT app.res
   ```

   For a Bourne shell, enter:

   ```
   XENVIRONMENT=app.res; export XENVIRONMENT
   ```

Note – There are other ways to get X to recognize your X resource file. To find out what they are, you will need to look them up in a book about the X Window System. See Appendix E, “Further Reading” for the names of some books you may wish to try.

---

10. **Run the application and verify that the menu behaves as expected.**
11. **Save the design and exit from Sun WorkShop Visual.**

CHAPTER 10

Designing for Java

Introduction

This chapter describes the way Sun WorkShop Visual can generate Java code with the option of using the Swing component set, from any design. It is divided into the following sections:

1. **Requirements**. This describes what you will need in order to use the generated Java code. This starts on page 315.
2. **Using Sun WorkShop Visual for Java**. This section describes how to use Sun WorkShop Visual to create a Java-compliant design. This section starts on page 315.
3. **Java Version**. This section describes how you can generate code for Java 1.0, 1.1 and Swing and how to tell which resources apply to which version. This starts on page 319.
4. **Widgets**. This introduces the way in which a Motif design can be generated in Java, starting on page 323.
5. **New Widgets for Java Classes**. This section describes the extra layout widgets for those Java layout components which have no Motif equivalent. See page 323.
6. **Event Model**. This describes how you can use the Java 1.1 event model, complete with *listener objects*, in Sun WorkShop Visual. A short tutorial is provided to help you get started. This section starts on page 330.
7. **Generate Dialog**. Changes have been made to the Generate dialog to allow you to generate Java code. This descriptions starts on page 336.
8. **Generated Code**. This section is a discussion of the code generated for Java and starts on page 339.

9. **Moving Sun WorkShop Visual Designs to Visaj®**. This section describes how you can take your Java design further by saving your design in a format which can be imported into Visaj, the visual application builder for Java. See page 347.

10. **Motif Widgets to Java Classes - the MWT Library**. This section describes the library of classes which map Motif widgets to Java classes. This starts on page 350.

What Is Java?

Java is a programming language with elements reminiscent of C and C++ (amongst others). It has libraries specifically geared for the Internet environment. In addition, Java is highly portable, object-oriented and interpreted. It is threaded, has automatic storage management and uses exceptions. If none of this means anything to you, you may like to read the books listed in “Books on Java” on page 887 before continuing.

Swing

Sun WorkShop Visual can also generate code to use the Swing component set. Swing components use the JFC (Java Foundation Classes) to give a set of components which are independent of the underlying window system. They also feature the “pluggable look and feel” built into the JFC. All of this means that you can create one user interface which can reflect the look and feel of any of the major Java platforms (Windows, Solaris, Macintosh). This look and feel can even be switched at runtime without the need to restart the application. Swing gives your application a consistent interface and platform independence.

The Generated Java Code

The code generated by Sun WorkShop Visual can be in the form of *applets* or straightforward applications. An applet is a small application which is embedded in a web page and runs when the page is browsed.

The generated Java code is not restricted in any way. That is to say, the code can be taken away and used on any platform supporting Java and used any number of times. Because Sun WorkShop Visual allows you to design on Motif-based platforms regardless of your target platform, the generated code imports a library of classes implementing the mapping of Motif widgets to Java classes. This library is known as *MWT* and is supplied as part of the Sun WorkShop Visual release. In Java, classes can be grouped together into *packages*. The MWT is in a package called `uk.co.ist.mwt`.

"Motif Widgets to Java Classes - the MWT Library" on page 350 details the way in which Motif widgets have been mapped to Java classes.

Requirements

In order to compile and run the generated Java code, you will need a Java compiler and interpreter. See your Sun WorkShop Visual Installation Notes for details on obtaining a Java compiler and interpreter. If you intend to generate applet code, you will also need either an applet viewer or a HTML browser to view the applet.

To generate Java code from Sun WorkShop Visual, you do not need to set any environment variables other than those you already use for Sun WorkShop Visual. In order to compile and run the generated code, however, you will need to set the CLASSPATH environment variable. This, like the PATH environment variable, is a list of directories. This list *must* include the directory $VISUROOT/lib/java_classes - where VISUROOT is the install directory of your Sun WorkShop Visual. It should also include any directory where you have generated class definitions which will be used by your application - for example, the current working directory (.). If you have generated Swing code from your design, your CLASSPATH must also include the Swing jar file. You should also make sure that the directory containing your Java compiler and interpreter is in your PATH list.

Using Sun WorkShop Visual for Java

The way Sun WorkShop Visual is used in order to generate Java code bears some resemblance to the way Sun WorkShop Visual is used for the generation of C++ and Microsoft Windows code. This is because Java is a class-based language like C++. The way in which class instantiations, derived classes and methods are handled is the same.

Note – If, therefore, you are not familiar with the use of Sun WorkShop Visual for C++, it is strongly recommended that you refer to Chapter 8 ”Structured Code Generation and Reusable Definitions“.

User-Defined Widgets and Java

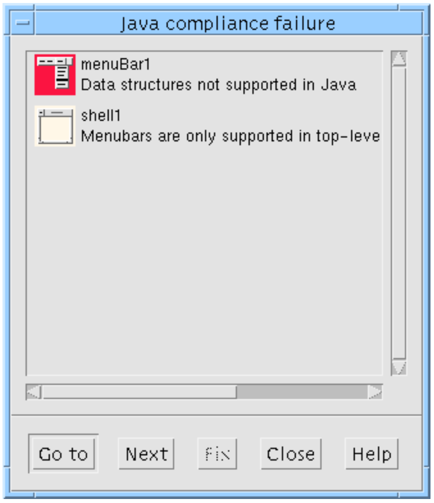
You can generate Java code for designs containing user-defined widgets. This is detailed in “Generating Java Code” on page 629 in Chapter 23 ”User-Defined Widgets“. Using the resource file mechanism, the widgets are mapped to Java components. The section mentioned above details how to do this and where to find existing resource files.

Creating Java Compliant Designs

The “Module” menu contains a toggle labelled “Java compliant”. Setting this toggle on indicates to Sun WorkShop Visual that you wish your design to be suitable for Java code generation. If your design cannot be reproduced in Java code, the Java Compliance Failure dialog is displayed.

Java Compliance Failure Dialog

The Java Compliance Failure Dialog lists all aspects of the design which cannot be reproduced in Java code. The dialog is illustrated in Figure 10-1.



menuBar1
Data structures not supported in Java

shell1
Menubars are only supported in top-leve

FIGURE 10-1 Java Compliance Failure Dialog

The buttons at the bottom of the dialog perform the following functions:

- **Go to**. Pressing this button causes the widget selected in the Java Compliance Failure Dialog to become selected in your design. If the widget is in a Shell other than the currently viewed one or it is in a folded section of the hierarchy, the view changes so that the widget can be seen.
- Next. Pressing this button moves the selection in the Java Compliance Failure Dialog to the next item in the list of offending widgets.
- Fix. Pressing this button fixes whatever is causing the compliance failure for the selected widget in the dialog. This button is only enabled when it is possible for the compliance failure to be fixed automatically in this way.
- Close. This button closes the dialog.
- Help. This button displays help on the dialog.

Design Restrictions for Java Code Generation

The following is a list of user interface features which cannot be carried over to Java code:

1. A file selection box with a parent which is not a shell.

   A file selection box must have a shell as a parent; Java only has a modal file selection dialog.

2. A file selection box which has a work area.

3. A file selection box which is a class.

4. A widget which is a class between a menubar and a shell.

   This restriction is imposed so that the shell can find out what its menubar is.

5. More than one menubar in a shell

   In Java, a shell may only have one menubar.

6. A cascade button has no menu.

   Because there is no equivalent to a cascade button in Java (menus are added directly to menubars), a cascade button on its own is meaningless.

7. The design contains a popup menu.

   Java does not support popup menus.

8. There is a multi-line string in a menubar.

   Multi-line strings are not supported in menubar. They are supported elsewhere.

9. A link source is not a class.

   The source of a link must be a class in Java.

10. A callback source is not within a class.

    This is a restriction of the current version of the AWT event model and may be improved in later versions as this model develops.

11. A pixmap in a menu.

12. A Label in a menu.

13. A menu in the main window of an applet

    The three items above are not supported in Java.

Applet Design Rules

There are some design restrictions which apply specifically to applets. These are:

1. An applet must have an application shell as the parent

2. The applet itself must *not* be a class, although its immediate child *must* be a class.

3. Any other shells should be top level shells.

4. All top level shells should be classes.

To Make a Class or Not To Make a Class...

The Java compiler applies a restriction to class methods. A class method must contain no more than 64 variables. This has a major impact on your design although this is not immediately apparent. 64 variables sounds a lot but it is easy to reach such a figure. Once you do hit this limit, you will need to start designating widgets as classes so that there are more methods, each with the 64 variable allowance. If your design does overstep this limit, Sun WorkShop Visual informs you when you generate Java code.

There is a shortcut method of making a widget a class; simply hold down the right mouse button to display a menu of useful commands, including “Make class”.

Enclosing Class for Callback Declarations

Widgets in Java designs can be given callbacks in the same way as widgets in any other type of design. Unless the callback is intended to define a listener object (as described in “Event Model” on page 330), the callback should be a method. The method is declared in the *enclosing class* for the widget. If the widget which has been given the callback is itself a class, then the callback method is declared in the widget’s class. If it is not, Sun WorkShop Visual searches up the hierarchy to find the nearest ancestor which is a class and declares the method in the class for that widget.

Java Version

By default, Sun WorkShop Visual supports Java version 1.1. You may generate code for Java 1.0 or for Swing by selecting the appropriate option in the generate options dialog. This is described in “Java Version for Generated Code” on page 319.

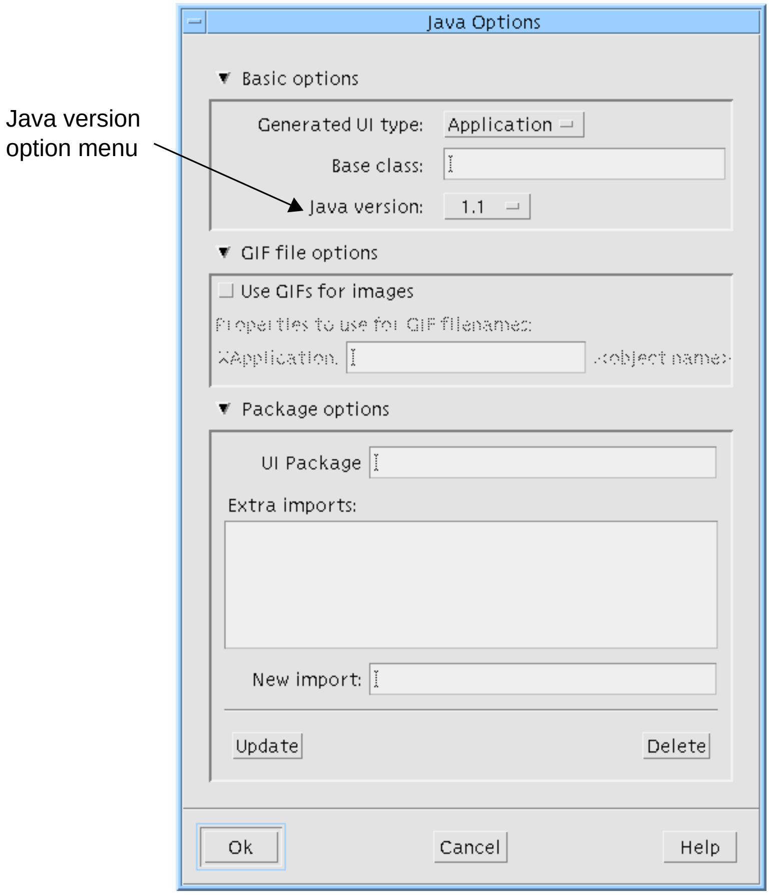
Resources and callbacks are marked to indicate which version (or versions) of Java supports them. A full description is provided in “Version Symbols for Resources and Callbacks” on page 321.

The event model (the way messages are passed between widgets) changed with Java 1.1. Sun WorkShop Visual fully supports this model by using *listener objects*. All of this is detailed in “Event Model” on page 330.

You can find out which version of Java you are using by running java with the “-version” switch. The version is then printed on standard output and Java immediately exits.

Java Version for Generated Code

The Java Options dialog, shown in Figure 10-2, includes a new option menu allowing you to select the Java version for your generated code.



Generated UI type: Application

Base class:

Java version: 1.1

□ Use GIFs for images

Properties to use for GIF filenames:

XApplication. <object name>

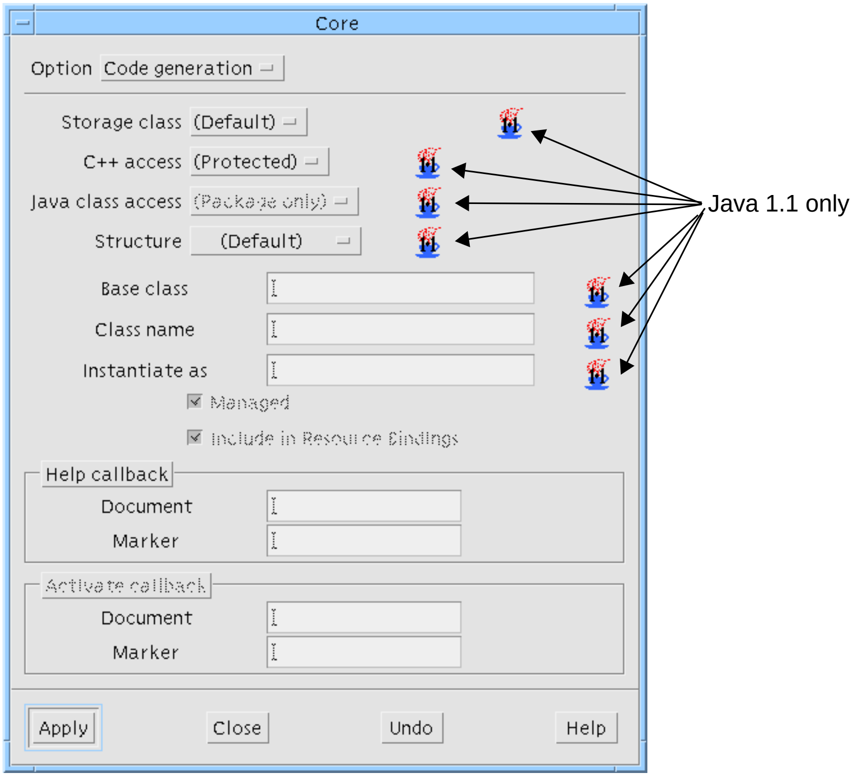
FIGURE 10-2 Java Options Dialog

The Java Options dialog is displayed by pressing the button labelled “Java options...” on the Java page of the Generate Overview dialog.

Whichever version was last selected is the version used when code is generated from the command line. If no selection has been made, the default of “Java 1.1” is used. If you choose “Swing” from this option menu, your generated application can run with any compatible version of Java including 1.1 and Java 2.

Version Symbols for Resources and Callbacks

There are four annotation symbols in the resource panels to show Java support. The coffee cup with the text “1.1” printed over it 1.1 , as shown in Figure 10-3, indicates that the resource is supported in Java 1.1 only.



Core

Option Code generation

Storage class (Default)

C++ access (Protected)

Java class access (Package only)

Structure (Default)

Base class

Class name

Instantiate as

☑ Managed

☑ Include in Resource Bindings

Help callback

Document

Marker

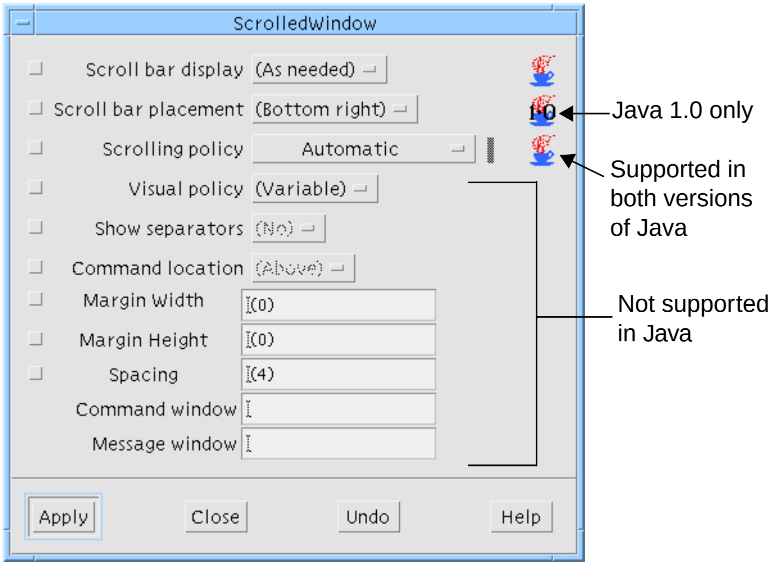
Activate callback

Document

Marker

FIGURE 10-3 Java 1.1 Core Resources for Popup Menu

The text “1.0” over the coffee cup 1.0 , shown in Figure 10-4, indicates that the resource is supported in Java 1.0 only. A coffee cup with no text ☕ indicates that the resource is supported in both versions of the JDK. The coffee cup with the letter “S” over it S indicates that the resource is supported in Swing. Having no coffee cup indicates that the resource has no Java support.



Scroll bar display (As needed)

Scroll bar placement (Bottom right)

Scrolling policy Automatic

Visual policy (Variable)

Show separators (No)

Command location (Above)

Margin Width (0)

Margin Height (0)

Spacing (4)

Command window

Message window

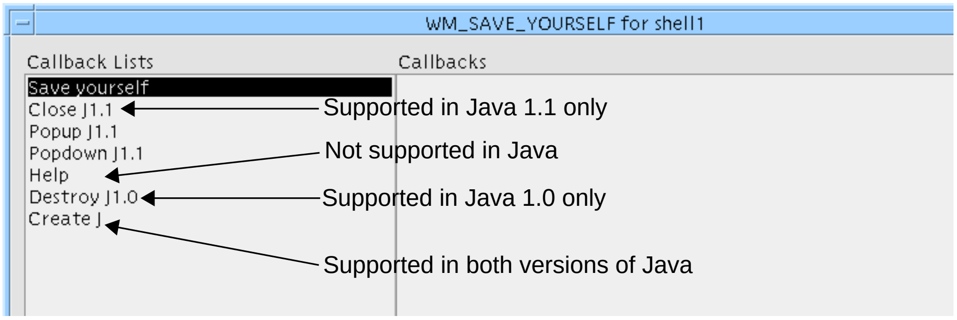
Java 1.0 only

Supported in both versions of Java

Not supported in Java

FIGURE 10-4 Resource Panel Annotations

Callbacks in the callbacks dialog are shown with the string “J1.0” to indicate that the callback is supported in Java 1.0 only. The string “J1.1” indicates that the callback is supported in Java 1.1 only. A simple “J” indicates that a callback is supported in both versions. As with resources, no annotation means that the callback is not carried forward into Java. Examples of these annotations are shown in Figure 10-5.



Save yourself
Close J1.1
Popup J1.1
Popdown J1.1
Help
Destroy J1.0
Create J

Supported in Java 1.1 only

Not supported in Java

Supported in Java 1.0 only

Supported in both versions of Java

FIGURE 10-5 Callback Dialog Annotations

Widgets

The set of Java components and the set of Motif widgets are not the same. This means that two areas need to be covered to resolve the problem of generating Java code from a design built on a Motif-based platform:

1. Map the Motif widgets available from Sun WorkShop Visual to Java classes
2. Add widgets to represent Java classes which cannot be mapped back to Motif widgets

In order to address the first issue listed above, the MWT is supplied with Sun WorkShop Visual. This is a library of classes which mimic the Motif widgets. This library is called *MWT* and is located in $VISUROOT/lib/java_classes. The classes in this library have been grouped together into a package which is referred to as `uk.co.ist.mwt`. All code generated from Sun WorkShop Visual imports this package.

Internally, Sun WorkShop Visual decides which widgets correspond to which Java classes. A full listing of this mapping is provided in “Motif Widgets to Java Classes - the MWT Library” on page 350 along with a list of the resources which are relevant to Java code.

To satisfy the second criterion above, the following Java layout classes, which have no counterpart in Motif, are provided as widgets:

- Card
- Flow
- Border
- Grid
- GridBag

These are detailed below.

New Widgets for Java Classes

There are five new widgets; each one simulates a Java layout manager. The widgets are a part of Sun WorkShop Visual and, as such, appear on the palette. The source code for them is also supplied as a part of the Sun WorkShop Visual release. This can be found in $VISUROOT/src/java_widgets (where VISUROOT is the install directory of your Sun WorkShop Visual). The directory also contains a file named README which provides more information on the widgets.

The following sub-sections provide information on how to use each of the new Java widgets and the resources associated with them.

The Card Widget

The Card widget is a Motif equivalent of the Java CardLayout class. It lays out its children so that they are all the same size as itself with only the topmost child visible. Each child can be given a “page number” and the Card widget can be told to show the child with a specified page number.

A typical use for the Card widget is multi-page dialogs controlled by an Option menu or by “Tab” buttons.

There are three resources associated with the Card widget:

- horizontalSpacing
- verticalSpacing
- currentPage

The spacing resources are of type XtRDimension and refer to the spacing around the Card.

The current page resource determines which of the various children of the Card is currently displayed “on top”. The resource is of type XtRInt.

Each child of a Card widget has a constraint resource, pageNumber, which allows you to assign a page number to the child widget. This resource is of type XtRShort.

To display any given child, set the Card widget currentPage resource to the pageNumber specified for the child.

There is a convenience function XdCardShowPage(Widget card, int page) defined in the Card widget sources for performing this within your own code.

The Flow Widget

The Flow Widget is a Motif equivalent of the Java FlowLayout class. It lays out its children in rows from left to right. When a row is full it moves onto a new row i.e. it lays out its children as a word processor lays out words in a paragraph. The rows may be left/right aligned or centred.

There are three resources associated with the Flow widget:

- horizontalSpacing
- verticalSpacing
- horizontalAlignment

The horizontal and vertical spacing resources are of type XtRDimension. They refer to the gap between the rows and columns of child widgets.

The horizontal alignment resource is of type XdRAlignment and can be set to one of left, center or right.

The resource behaves similarly to the alignment of text in a paragraph.

The Border Widget

The Border widget is a Motif equivalent of the Java BorderLayout class. It can accept up to five children, which can be assigned to five positions: north, south, east, west and center, as shown in Figure 10-6. The Border widget can have more than five children but the surplus ones are simply added at a default location and not laid out.

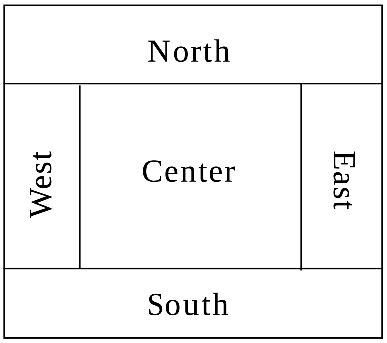


FIGURE 10-6 Schematic Depiction of BorderLayout

The north and south children expand to fill the width of the Border widget. The west and east children fill the space between the north and south children vertically and the center child fills any space left over. The Border widget makes a good replacement for the MainWindow widget as it does not suffer from many of the MainWindow's problems and peculiarities. Any of the five positions may be left empty.

There are two resources for the Border widget:

- horizontalSpacing
- verticalSpacing

Both of these resources are of type XtRDimension and refer to the gap between the children.

Each child of the Border widget has a constraint resource which is the borderAlignment. This is of type XdRBorderAlignment, and can be set to north, south, east, west, or center.

The Grid Widget

The Grid widget is a Motif equivalent of the Java GridLayout class. The Grid widget makes all its children the same size and lays them out in a grid pattern. The number of columns in the grid is specified by a resource - the number of rows is calculated. Resizing the Grid widget will cause all its children to resize to fit.

There are three resources for the Grid widget:

- horizontalSpacing
- verticalSpacing
- numColumns

The horizontal and vertical spacing resources are of type XtRDimension and refer to the gap between the rows and columns of the children.

The numColumns resource is of type XmRShort, and this refers to the number of columns to display in the grid.

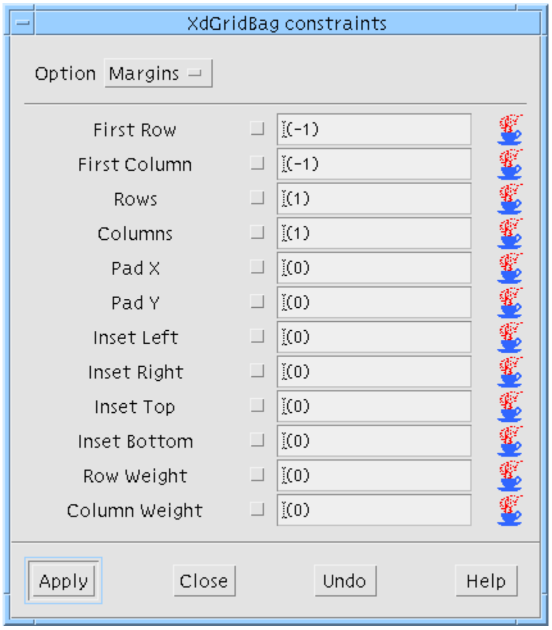
The GridBag Widget

The GridBag widget is a Motif equivalent of the Java GridBagLayout class. The GridBag widget is complicated but provides the greatest flexibility of all the Motif and Java layout widgets. The GridBag incorporates the idea of a grid - with widgets in cells laid out in rows and columns. The rows and columns of the GridBag, unlike those of the Grid widget, can be different heights and widths to accommodate the different types of widget. The size of a row or column is determined by the tallest or widest child widget in that row or column respectively. Child widgets can also be expanded to fill any number of rows and columns and can be aligned to a specified geographical location within the cell(s) that they occupy. The GridBag also allows you some control over the way children resize when the GridBag itself is resized. This is explained in “Resizing the GridBag” on page 328.

When children are added to the GridBag, they are placed to the right of the previously added widget. To change their position in the grid, you will need to use the constraints dialog on each individual child widget.

GridBag Constraints Dialog

The Constraints dialog for the child of a GridBag is shown in Figure 10-7.



First Row ☐ (-1)
First Column ☐ (-1)
Rows ☐ (1)
Columns ☐ (1)
Pad X ☐ (0)
Pad Y ☐ (0)
Inset Left ☐ (0)
Inset Right ☐ (0)
Inset Top ☐ (0)
Inset Bottom ☐ (0)
Row Weight ☐ (0)
Column Weight ☐ (0)

FIGURE 10-7 Constraints Dialog for GridBag

The top of the Constraints dialog shows two option menus. The cellAlignment resource tells the GridBag where the widget should be placed geographically in the group of cells that it occupies - North, South, East, West, NorthEast, NorthWest, SouthEast, SouthWest or Center. The cellFill resource makes the widget resize to fill part or all of the group of cells that it occupies. Selecting Horizontal makes the widget fill all the columns that it occupies, selecting Vertical makes it fill all the rows. Selecting Both makes the widget expand to fill all the rows *and* columns that it has been given. The None option keeps the widget at its original size at the geographical location specified by the cellAlignment resource.

The row and column resources allow you to tell the GridBag where you wish the selected widget to appear in the grid. The similarly sounding rows and columns resources tell the GridBag *how many* rows and columns the widget will occupy. The widget does not expand to fill the specified number of rows and columns - that behavior is controlled by the cellFill resource described below. There are two “special” values which can be entered in the rows and columns textbox:

1. A value of 0 (zero) in the columns or rows textbox indicates that the widget should occupy all columns or rows respectively from its current position to the edge.

2. A value of -1 (minus one) in the column or row textbox indicates that the widget should remain vertically or horizontally next to the previously added widget, respectively.

The rowWeight and columnWeight resources refer to the resize policy and are explained in “Resizing the GridBag” on page 328.

The padX and padY resources specify internal padding to add on each side of the component horizontally and vertically respectively.

The inset resources (insetLeft, insetRight, insetTop and insetBottom) specify the margins to appear on each side of the selected widget.

Resizing the GridBag

The Constraints dialog for the children of a GridBag contains the rowWeight and columnWeight resources. These resources affect the way the rows and columns will resize when the GridBag is resized.

Although each widget child of a GridBag can be given a rowWeight and a columnWeight, the GridBag searches for the highest number in each row and column and uses that number for its calculations for the whole row or column. For this reason there is no need to set a weight on *every* widget, just one in each row and column.

The easiest way to describe the effect of setting row and column weights is through examples.

rowWeight Example

This first illustration considers rowWeights. Imagine that you have set row weights as shown in Figure 10-8. The top row has a highest rowWeight setting of 3, the middle 2 and the bottom row 1. These “highest” settings are the only ones that matter - we can now ignore the other rowWeights in each row.

The GridBag now calculates the sum of all rowWeights - in our case, this is 6. Now imagine that the GridBag is stretched downwards. Any extra space is allocated to the rows as follows:

- The row with a rowWeight of 3 receives one half of the extra space (because 3 is one half of 6)
- The row with a rowWeight of 2 receives one third of the extra space (because 2 is one third of 6)
- The last row, with a rowWeight of 1 receives the remaining one sixth of the extra space (because 1 is one sixth of 6)

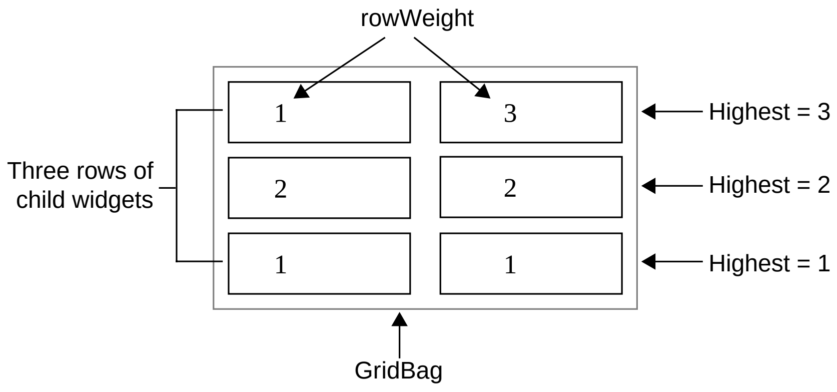


FIGURE 10-8 Example rowWeight Settings

columnWeight Example

The GridBag calculates how to resize its columns in the same way as it calculates the rows, explained above. Imagine a GridBag containing widgets which have been given columnWeights as illustrated in Figure 10-9. This is the same illustration as that shown in Figure 10-8 except that the figures now refer to columnWeight instead of rowWeight. The highest columnWeight value in the left column is 2 and the highest value in the column on the right is 3. The GridBag is only interested in the highest figure in each column and ignores any other columnWeights that you may have set. The sum of these "highest" columnWeights is 5. With all of these figures, the GridBag can perform some percentage calculations when there is extra space available.

Now imagine that the GridBag depicted is stretched outwards to the right. It allocates the extra space to its columns as follows:

- The left column, with a columnWeight of 2 receives two-fifths of the extra space (because 2 is two-fifths of 5)
- The right column, with a columnWeight of 3 receives the remaining three-fifths of the extra space (because 3 is three-fifths of 5)

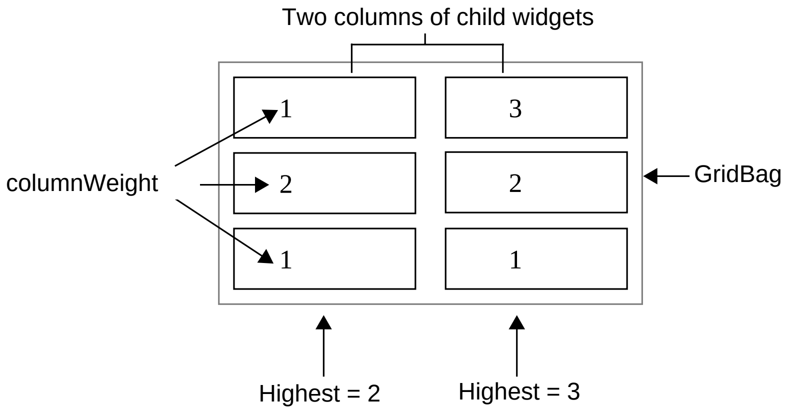


FIGURE 10-9 Example columnWeight Settings

GridBag Resources

There are two resources for the GridBag widget:

- horizontalSpacing
- verticalSpacing

Both of these resources are of type XmRInt and refer to the gap between the rows and columns.

Event Model

The event model (the way messages are passed between objects) changed between versions 1.0 and 1.1 of Java. If you select “Java 1.1” as the version to generate, the new event handling mechanism is used.

Whichever mechanism is being used, if you have registered a callback *method* (as opposed to a *function*), your method is called when the event is triggered. The main difference between the two versions is in the way callback *functions* are handled.

If you generate Java 1.0 code, all callback *functions* are ignored. They are not generated. Java deals with classes and therefore only supports the concept of *methods*.

When generating Java 1.1 code, however, your callback *functions* are treated as calls to an external listener object which can be generated by Sun WorkShop Visual. This means that if you are moving from C++ to Java and your code uses functions, you can generate Java 1.1 code and maintain equivalent behavior.

Note – The buttons in a Java File Selection Box do not fire events when pressed. For this reason, links from a File Selection Box will not work in the generated Java code.

Listener Objects in Sun WorkShop Visual

When generating code for Java 1.1, Sun WorkShop Visual interprets callbacks which have been defined as functions (as opposed to methods) as *listener objects*. A listener object is an instance of a class which implements an interface between an action occurring in the user interface front end and the rest of the application. These are also called "helper" and "adapter" classes.

Sun WorkShop Visual can follow one of two strategies when generating listener objects from callback functions. This is controlled by the following resource:

visu*javaWrapFunctions:true

The default is "true". This causes Sun WorkShop Visual to generate a wrapper object, as explained in the following subsection. Set this resource to "false" if you wish to handle the listener object code yourself, as described in "Controlling Listener Objects Yourself" on page 333.

Using Callback Objects

When Java 1.1 is generated, callback functions are treated as references to listener objects. A callback object is generated into its own file using the name of the function with "callback" appended. If, for example, you have a callback function named "myFunction", the following class is generated in a separate file called `myFunctionCallback.java`:

```
public class myFunctionCallback
{
        static myFunctionCallback me = null;

        public void doit( AWTEvent e, Object context)
        {
          // write your code here
        }
        static myFunctionCallback getInstance()
        {
          return new myFunctionCallback();
        }

        static myFunctionCallback get()
        {
          if (me == null)
                 me = getInstance();
          return me;
        }
}
```

The file is only generated if it does not already exist. You can safely add to it because this file is not overwritten.

There is only one instance of each callback's object. If you have added the same callback function to a number of different widgets, they will share the one callback object by calling a get() method which returns the single instance. The object is created the first time get() is called.

The callback object has a doit() method which is similar to a standard callback stub - this is where you add your own code. The doit() method is called from the listener object in the enclosing class of the object with the defined function. The context object and the event itself are passed into the doit() method. For example:

```
{ // Java 'global' callback object
     final myFunctionCallback myFunctionHandler =
           myFunctionCallback.get();
     if ( myFunctionHandler != null )
           button1.addActionListener(
                  new ActionListener()  {
                      public void actionPerformed(ActionEvent event)
                      {
                         myFunctionHandler.doit(event,
                             XApplication.this);
                      }
                  }
           );
}
```

This is the default behavior. Changing the javaWrapFunctions resource to “false” results in the behavior described in the following section.

Controlling Listener Objects Yourself

If the javaWrapFunctions resource is set to “false”, none of the callback object code described above is generated. In the example of the Activate callback on a button, the following is generated into the enclosing class:

```
if (myFunction != null)
     button1.addActionListener( myFunction );
```

Where “button1” is the name of the widget which has a callback function defined for it and “`myFunction`” is the name of the callback function.

The code as generated is intentionally incomplete; it will not compile as it is. There are two pieces of code which you need to add:

1. The declaration of the listener object “myFunction”.

2. The definition of the class of which “myFunction” is an instance.

The first piece of code is relatively simple. It would look like this:

```
EventListener myFunction = new EventListener();
```

This declares the listener object “`myFunction`” as an instance of the class “EventListener” which is a class that you are about to define.

The second piece of code is a little more complex:

```
class EventListener implements TextListener, ActionListener {
public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == button1)
            // Add the code to do something here...
    }
}
```

This is the definition of the new EventListener class. The method “actionPerformed” is called when an event occurs in button1. You may wish to define classes like this in their own file with “public” access so that they may be used by any component. You will then need to make sure that you can access the component (in this case “button1”).

Note – Only callbacks which are marked as Java compliant in the Callbacks dialog are generated. Those without the Java mark in the Callback List are ignored when Java code is generated.

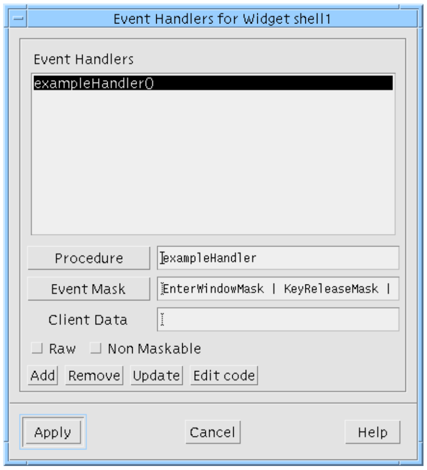
X Events as Listener Objects

Selecting “Event Handlers...” from the Widget menu, when you have a widget selected, displays the Event Handlers dialog which is described in “Event Handlers” on page 203. Any procedures added here are treated the same as callback functions when Java code is generated *if you have specified Java-compliant event masks*. Event masks which are not Java-compliant are ignored when Java code is generated.

Sun WorkShop Visual assumes that you have a listener object with the name given for the procedure, in the same way it does for callback functions. Sun WorkShop Visual registers the listener object for each Java event type corresponding to each X event mask. For example, Figure 10-10 shows an event handler with the following event mask:

```
PointerMotionMask | KeyReleaseMask | EnterWindowMask
```

for a procedure called “exampleHandler” on a widget named “MyButton”.



exampleHandler()

Procedure: exampleHandler

Event Mask: EnterWindowMask | KeyReleaseMask |

FIGURE 10-10 Example Event Handler

For such an event handler, Sun WorkShop Visual would generate the following Java code:

```
if ( MyHandler != null )
    MyButton.addMouseMotionListener( myHandler );
```

```
if ( MyHandler != null )
    MyButton.addKeyListener( myHandler );
if ( MyHandler != null )
    MyButton.addMouseListener( myHandler );
```

You would then be expected to add a line to the generated code file declaring that myHandler is an instance of a class defined elsewhere, for example:

```
MyHandler myHandler = new MyHandler();
```

To compile this code, you would need to define the class MyHandler. This class would have the following signature (it could also be "public" if so required):

```
class MyHandler implements MouseMotionListener, KeyListener,
                                    MouseListener {
```

The MyHandler class would contain definitions of all the methods of the three listener classes.

Version Incompatibility for Callback Method Signatures

Callback methods for Java 1.0 have the signature:

```
void foo( Event x )
```

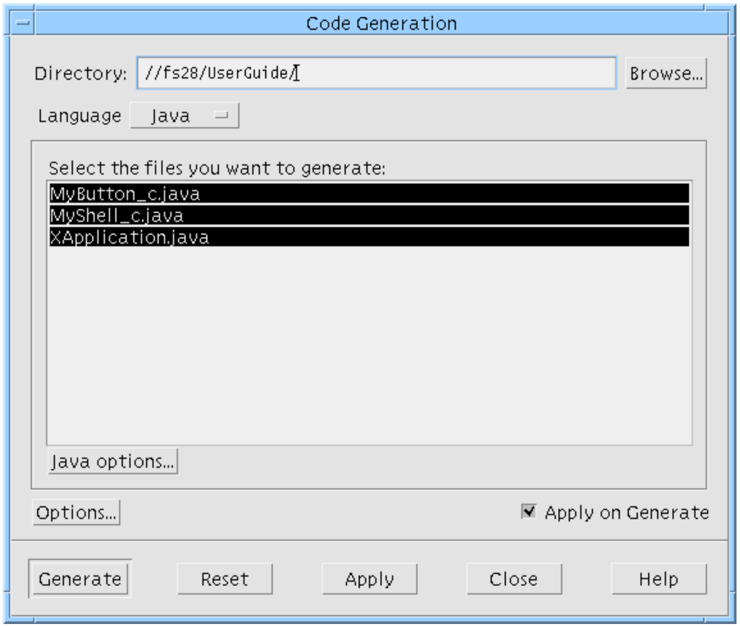
and for Java 1.1, they have the signature:

```
void foo( AWTEvent x )
```

The result of this is that if you generate a new Java 1.1 file on top of an existing Java 1.0 one, new stubs are generated for all of your callback methods. The old ones are, of course, retained, but they are no longer the methods which will be called. When upgrading your design from Java 1.0 to Java 1.1, you should move any code from the old methods to the new, adapting any code which uses the event objects.

Generate Dialog

The Generate Dialog is quite different when “Java” is selected from the Language option menu. A list of files which can be generated is displayed, as shown in Figure 10-11.



Code Generation

Directory: //fs28/UserGuide/

Language Java

Select the files you want to generate:

MyButton_c.java
MyShell_c.java
XApplication.java

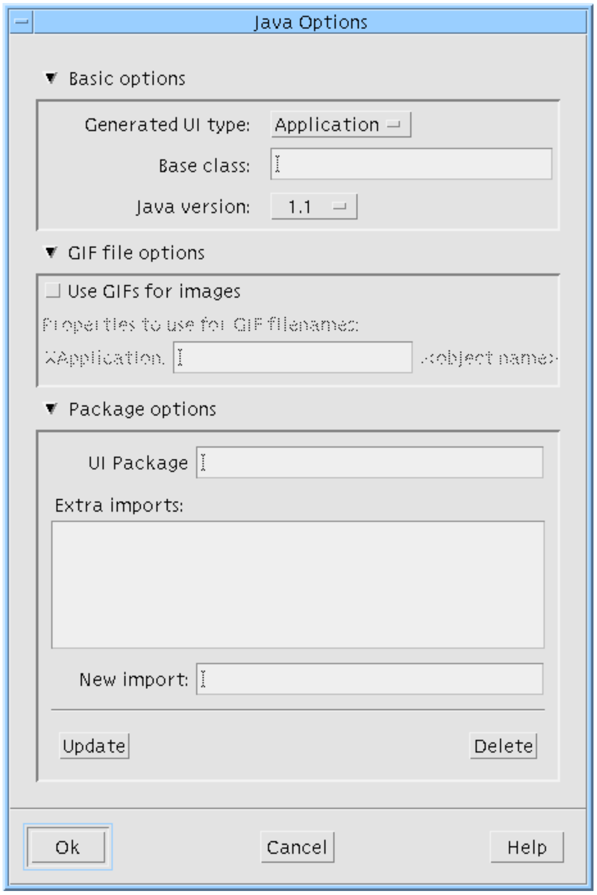
☑ Apply on Generate

FIGURE 10-11 Java Generation Dialog

The text box labelled “Directory” shows where the files will be generated. To change this either type the name of the directory directly into the text box or press the “Browse” button and use the File Selection dialog.

Only files which are selected (or highlighted) will be generated. To select or deselect a file, click over it. Use the Shift key modifier to extend the selection list and the Control key modifier to add individual files to the selection. For more details on the files listed in this dialog, see “Java Files” on page 340. “Using the Generated Files” on page 345 explains how to compile and run the generated code.

The button labelled “Java Options” displays a dialog containing options which are relevant to Java code generation, as shown in Figure 10-12. For a description of the Code Options dialog, which is produced when the button labelled “Options” is pressed, see “Code Generation Options” on page 218.



Java Options

▼ Basic options

Generated UI type: Application

Base class:

Java version: 1.1

▼ GIF file options

□ Use GIFs for images

Properties to use for GIF filenames:

XApplication. <object name>

▼ Package options

UI Package

Extra imports:

New import:

FIGURE 10-12 Java Generation Options Dialog

Java Generation Options Dialog

The Java Generation Options Dialog allows you to control the following five areas associated with Java code generation:

1. Whether you wish to create an application or an applet.
2. The name of the base class of the application.
3. Which version of Java code you wish generated - including the option of generating Swing.
4. Whether you wish your design to use GIFs or not.

5. The name of the package to create, if you wish your classes to be bundled together into a package.

Application or Applet

By default, Sun WorkShop Visual will generate an application from your design. If you prefer to generate an applet, select the appropriate option from the option menu. If you select “Applet”, the generated main code file will contain extra code to allow the application to run inside an HTML browser.

Changing the Base Class

The main code file contains one class which, effectively, links together your design. The name of this class is whatever you have specified as the application class in the Code Options dialog (or XApplication by default). This can be *extended* from another class by typing its name in the text box labelled “Base class”. If you are generating an application and you do not specify a base class, the application class is not extended. If you are generating an applet and you have not specified a base class, the Java class “Applet” is used by default.

Using GIFs

The GIF options section of the dialog takes on one of two forms, depending on whether you are generating an application or an applet. In both cases, there is a toggle labelled “Use GIFs for images” at the top of the section. If this toggle is not selected, the rest of the section is insensitive. When it is selected and you are generating an application, the code generated for pixmap objects assumes that all of the pixmaps are stored in separate GIF file. When the “Use GIFs for images” toggle is not set, the pixmap objects are stored in a file named `<ApplicationClassName>PixmapObjects.java` as arrays of integers. When the “Use GIFs for images” toggle is set, that file contains code to load the GIF files. Also in this case, an extra file is added to the list of Java code files. This is the property file and is named `<ApplicationClassName>Properties`. In this file, the GIFs are referenced as follows:

<ApplicationClassName>.<ApplicationImagesName>.<PixmapObjectName>

where “ApplicationImagesName” is the value entered into the text field in the GIFs options section of this dialog. For example, if you are using the default application class name, XApplication, and you have created a pixmap object named “TreeIcon” and you have entered the value “MyApplicationImages” into the text field, then the property name for the TreeIcon pixmap would be:

```
XApplication.MyApplicationImages.TreeIcon
```

and if you had another pixmap object, this time named “LeafIcon”, the property name would be:

```
XApplication.MyApplicationImages.LeafIcon
```

If you are generating an applet and you have set the “Use GIFs for images” toggle, the GIFs section of the dialog contains a text box which allows you to specify a directory (relative to the document base) where the GIF images can be found.

If you wish to use GIFs in your application or applet, you will need to create the GIF files. Sun WorkShop Visual provides some help by allowing you to generate XPM files for all your pixmap objects in one go. To do this, go to the “Pixmaps” page of the Generate Dialog. This contains a list of the pixmap objects in your design. Pressing “Generate” creates a XPM file for each object giving it the name `<PixmapObjectName>.xpm`. You can choose whether you wish to generate XPM2 or XPM3 format. Once you have generated the XPM files, you will need to convert them to GIF using a conversion utility. There are a number of freely distributed conversion utilities available including *pbmplus* and *netpbm*.

Specifying a Package

The last section of the Java Generation Options Dialog contains a text area where you can specify the name of a package. A package is a group of classes that are bundled together. If you choose to generate your classes as a package, your main code file will contain a statement to import it.

Note – There are some conventions which should be followed in the naming and organizing of packages. Your Java documentation will have more details on this.

Generated Code

Java code differs substantially from C or C++ code in its structure. All code in Java must be contained within a class. Each separate class must be contained within its own file. The “main” procedure is a method of the application class. The Java interpreter will locate this method and start the application.

Java Files

In the generate dialog, when “Java” is selected from the menu of language flavors, a selectable list of files is shown. Depending on the structure of your design, these will be as follows (names in angle brackets indicate a variable name according to the names you have used in the design):

1. `<ApplicationClassName>.java`. This is the principal code file for the application which is a class defining the application. One method of this class is the “main” procedure. The name of the file is determined by the name of the application class - by default “XApplication”. You can change this in the Code Options Dialog.
2. `<ApplicationClassName>Links.java`. This is the file containing code to implement the dynamic links between widgets. You do not need this file if you have not made any links.
3. `<ApplicationClassName>PixmapObjects.java`. If you have specified any pixmap objects, they will appear in a separate file.
4. `<ApplicationClassName>FontObjects.java`. If you have specified any font objects, they will appear in a separate file.
5. `<ApplicationClassName>ColourObjects.java`. If you have specified any colour objects, they will appear in a separate file.
6. `<ApplicationClassName>StringObjects.java`. If you have specified any string objects, they will appear in a separate file.
7. `<WidgetClassName>_c.java`. Every widget you have designated as a Java class (from its Core Resource Panel), must have its own code file containing the class definition. There will, therefore, be as many of these files as there are classes in your design. Unless you have a good reason for ignoring one of the widgets in your design, you will always need to generate these files.

Command Line Code Generation

For generating Java code from the command line, an extra flag has been added to Sun WorkShop Visual. The flag is ‘-J’.

Example Code

The code generated by Sun WorkShop Visual for the hierarchy and design shown in Figure 10-13 is listed below. To generate the code yourself, follow these steps:

1. **Select a Shell widget.**
2. **Name the Shell “MyShell”.**
3. **Make “MyShell” an Application Shell.**

   This is done in the Shell resource panel.
4. **Add the DialogTemplate to the Shell.**
5. **Name the DialogTemplate “MyMessageBox”.**
6. **Select three PushButtons and a Form widget.**
7. **Select a ScrolledText as the child of the Form.**
8. **Name the ScrolledText “MyScrolledText”.**
9. **Select the ScrolledText , press the right mouse button and select “Make class” from the menu which is displayed.**

   This is a quick way of changing the structure of the selected widget. Alternatively, you could display the Core Resource panel, go to the “Code Generation” page and change the “Structure” option menu to “C++/Java class”.

   You should now have the hierarchy shown in Figure 10-13.

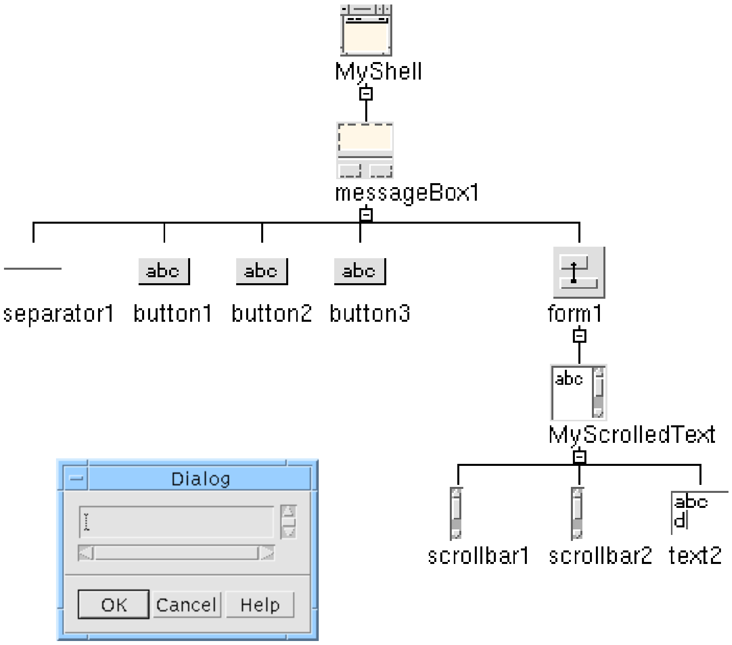
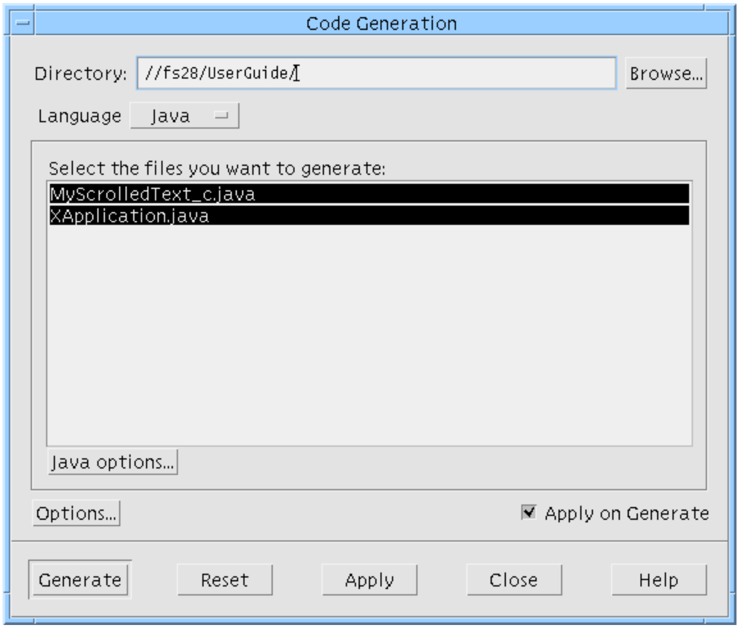


FIGURE 10-13 Hierarchy and Design for Example Code

This simple design has no resources set as we are only interested in the code which is generated from it. In the generate dialog, when “Java” is selected, we are offered the files shown in Figure 10-14.



Code Generation

Directory: //fs28/UserGuide/

Language Java

Select the files you want to generate:

MyScrolledText_c.java
XApplication.java

☑ Apply on Generate

FIGURE 10-14 Generate Dialog for Example Code

Because we have not specified an application class name, the default (XApplication) is used in the filenames. You will need the file XApplication.java as this is the main entry point for the application and you will also need the file MyScrolledText_c.java as this is the class definition of the ScrolledText widget. So, what you should do breaks down as follows:

10. **Make sure that XApplication.java and MyScrolledText_c.java are both selected.**

11. **Check the “Directory” shown at the top of the dialog. The files will be generated there so change it as necessary.**

12. **Press “Generate”.**

For important information on when to make widgets into classes see “To Make a Class or Not To Make a Class...” on page 318.

Below is a listing of the file `XApplication.java`:

```
/*
** Generated by Sun WorkShop Visual/Java(tm) Edition
*/
```

```
import java.awt.*;
import uk.co.ist.mwt.*;
/*
** Sun WorkShop Visual/Java(tm) Edition - main program
*/
/**
 * A class representing the user interface specified by the entire
 * design.
*/
public class XApplication  {
    protected MyScrolledText_c MyScrolledText;
    protected Panel MyMessageBox;
    DlogTemplateLayout MyMessageBoxLayout = new
          DlogTemplateLayout();
    protected Frame MyShell;
    public XApplication()  {
          Button button3;
          Button button4;
          Button button5;
          Panel form2;
          FormLayoutManager form2Layout = new
                 FormLayoutManager();
          MyShell = new Frame();
          MyMessageBox = new Panel();
          MyShell.add( “Center”, MyMessageBox );
          MyMessageBox.setLayout( MyMessageBoxLayout );
          button3 = new Button( “button3” );
          MyMessageBox.add( button3 );
          button4 = new Button( “button4” );
          MyMessageBox.add( button4 );
          button5 = new Button( “button5” );
          MyMessageBox.add( button5 );
          form2 = new Panel();
          MyMessageBox.add( form2 );
          form2.setLayout( form2Layout );
          MyScrolledText = new MyScrolledText_c();
          form2.add( MyScrolledText );
          FormLayoutConstraints MyScrolledTextConstraints =
                     new FormLayoutConstraints();
          form2Layout.constrain( MyScrolledText,
                     MyScrolledTextConstraints );
          MyShell.pack();
          MyShell.layout();
          MyMessageBox.layout();
          form2.layout();
          MyScrolledText.layout();
          MyShell.show();

        }

        public static void main( String args[] )  {
          new XApplication();
        }

}  /* ...class XApplication (main application class) */
/*
** (end of Sun WorkShop Visual/Java(tm) Edition generated main
program)
*/
/*
** (end of Sun WorkShop Visual/Java(tm) Edition generated code)
*/
```

Now, this is `MyScrolledText_c.java`:

```
/*
** Generated by Sun WorkShop Visual/Java(tm) Edition
*/
import java.awt.*;
import uk.co.ist.mwt.*;
/**
 * A class represented by the widget MyScrolledText_c in the design
file
 */
```

```
// Private: classHeader Sun WorkShop Visual-generated code - do not
edit >>>
class MyScrolledText_c extends TextArea  {
// Private: classHeader <<< Sun WorkShop Visual-generated code ends.


// Private: instanceVars Sun WorkShop Visual-generated code-do not
edit >>>
// Private: instanceVars <<< Sun WorkShop Visual-generated code ends.
/**
* The constructor method for MyScrolledText_c
*/
// Private: constructor Sun WorkShop Visual-generated code - do not
edit >>>
     public MyScrolledText_c()  {


     }
     // Private: constructor <<< Sun WorkShop Visual-generated code
ends.
// Private: endClass Sun WorkShop Visual-generated code - do not edit
>>>
}
// Private: endClass <<< Sun WorkShop Visual-generated code ends.
/*
** (end of Sun WorkShop Visual/Java(tm) Edition generated code)
*/
```

These files can be taken away and used in any way you see fit. The only caveat is the inclusion of the MWT. The line:

```
import uk.co.ist.mwt.*;
```

imports into your Java program all of the Java class library provided with Sun WorkShop Visual. This means that if you wish to take the generated files to a different platform and run the resulting application there, you will need to take the Java class library, MWT, as well.

Using the Generated Files

Once you have your Java code files you are free to use it on any platform which supports Java *as long as you have the MWT library on that platform*. Follow these steps to build a Java application from your generated files:

1. **Make sure that your CLASSPATH environment variable is set as explained in "Requirements" on page 315.**
2. **Make sure that the directory containing any classes you have just generated is included in the CLASSPATH list.**

   This may be your current working directory, in which case you should check that . (dot) is in the CLASSPATH list.
3. **Make sure that your PATH environment variable includes the directory containing your Java compiler.**
4. **Type:** `javac <ApplicationName>.java`

   This will give you a file <ApplicationName> with a ".class" extension.
5. **Type:** `java <ApplicationName>`

   Your Java application should appear on the screen.

Callback Stubs

All code in a Java code file must appear within a class. This means that the callback stubs are generated as part of the class to which they belong. There is no separate callback file as there is with other flavors. This is not a problem, however, because all code added to the generated code is retained *as long as the new code is outside of the guard comments*. This subject is explained in the following section, "Retaining Edits" on page 346.

Retaining Edits

The sample code above contains the following section:

```
/**
* The constructor method for MyScrolledText_c
*/
// Private: constructor Sun WorkShop Visual-generated code - do not
edit >>>
    public MyScrolledText_c()  {

    }
// Private: constructor <<< Sun WorkShop Visual-generated code ends.
```

This section contains a number of comments and illustrates how comments are used to retain any code you add. The first comment is for your information only. The second and third, however, are special. Any comment which begins "// Private: ..." - do not edit" indicates that the following lines must be left as they are. There is a corresponding "//Private: ... ends" which indicates the end of a section to be left untouched.

Any code or comment that you add *outside* of the special comments detailed above, will be retained when the file is regenerated. This means, for example, that you can add code to callback methods which will always remain.

Javadoc

Along with the Java compiler and interpreter you also have javadoc, an automatic documentation tool. Javadoc looks at .java files, parses the declarations and comments beginning with "/**" and produces HTML pages detailing the chain of class inheritance and all the public fields in a class along with any extra information contained in the special comments. The special comment mechanism is there for you to add to the documentation that javadoc creates. Code generated by Sun WorkShop Visual includes some basic information for Javadoc.

Moving Sun WorkShop Visual Designs to Visaj

Sun WorkShop Visual allows you to import your Sun WorkShop Visual designs into Visaj, the Java development tool. This gives you the ability to move your legacy Motif C/C++ designs quickly to Java.

You can, of course, use Sun WorkShop Visual's Java code generation facility to convert your design to Java. This would allow you to maintain a common code base covering Java, C/C++ for Motif and MFC for Microsoft Windows. If, however, you intend to use Java only, you may wish to move on to Visaj, IST's visual application builder for Java.

Moving your design from Sun WorkShop Visual to Visaj is simple and involves only the following steps:

- Save the design in a special format using Sun WorkShop Visual's Save As dialog.
- Load the file using Visaj's Import dialog.

This section describes how to achieve this from both sides - Sun WorkShop Visual and Visaj. The section ends with some troubleshooting hints.

From Sun WorkShop Visual

To save a file for Visaj, select ‘Save as...’ from the File menu and then select ‘Visaj bridge format’ from the ‘Save format’ option menu.

Specify a filename and press “OK”. This name will **not** be used as the default filename for your design.

If the design is *not* Java 1.1-compliant, the compliance dialog appears. This is the same dialog displayed when you generate Java form a non-compliant design. Fix the errors and try saving again.

As the file is generated, windows may flash up on the screen.

To Visaj

The Import pullright menu in the File menu contains the item “X-Designer bridge file” (X-Designer and Workshop Visual save files are fully compatible). When this is selected, a file dialog appears. Type the name of the file saved from Sun WorkShop Visual.

If you have used a feature in Sun WorkShop Visual which has no equivalent in the version of Visaj which you are using[1], a dialog is displayed describing what has been found. The possible items in this dialog are as follows:

One or more classes were expanded into simple component hierarchies

In Sun WorkShop Visual, you can make any component a class. In Visaj, classes are separate hierarchies in different designs. To use them together in a design:

1. **Generate code for each class.**

2. **Compile the code.**

3. **Add the compiled classes to the palette as beans.**

4. **Use the newly added beans in a larger design.**

This method of conversion from Sun WorkShop Visual to Visaj allows you greater control of the way classes are used than if Sun WorkShop Visual made those decisions for you.

[1] This applies to version 2.0 of Visaj.

While importing, Visaj “expands” classes. You can, of course, refine this by cutting subhierarchies and pasting them into new Visaj designs after importing the design.

One or more frames were moved into their own classes

Visaj supports one frame per class. If your design contains multiple application shells or top-level shells, it is converted into several Visaj class files - one for each shell.

Dialogs, by contrast, are all reparented to the application shell (or a top-level shell if there is no application shell). However, if there is no application shell and there are no top-level shells, but if there are multiple dialogs, then each dialog is placed into its own class.

String/font/color objects in your design were expanded into simple property settings

Visaj has no equivalent to these objects, so all references to objects are expanded into simple resource values. For example, if you have a label myLabel whose text is <myObject>, where <myObject> is a string object with the value “Hello”, then after import, myLabel has the text “Hello”, and <myObject> is no longer used.

All XmForms in the design were changed to absolute positioning using a null layout

In Visaj, there is no layout manager `com.pacist.mwt.FormLayoutManager`. Components whose layout is controlled by this manager are positioned absolutely as a result.

Note – You are recommended to change your design to use one of the Java layout managers.

After Import

Once imported into Visaj, your design reflects exactly the Motif design in Sun WorkShop Visual. This is the same as if you had generated Java code directly except that, with Visaj, you can continue your Java application development.

Motif Widgets to Java Classes - the MWT Library

You may design your application user interface using Sun WorkShop Visual on a Motif-based application and then generate both C/C++ code for Motif and Java code. Sun WorkShop Visual allows you to do this by providing a library of Java classes, called the MWT, which map Motif widgets to Java classes. Some Motif widgets have a direct counterpart in Java, others do not. Where there is no clear mapping, the MWT library attempts to mimic the Motif widget using available Java classes. The following table lists the Motif widgets available on Sun WorkShop Visual's widget palette and show which Java or MWT class is being used in the Java code generated.

Note that if you are generating Swing, Sun WorkShop Visual tries to map the Motif widgets to Swing components and if it cannot, uses a separate MWT library specifically for Swing components. This is detailed in the following section, "Mapping Motif Widgets to Swing Components" on page 353.

TABLE 10-1 Mapping of Motif Widgets to Java Components

Motif Widget	Java Component
Shell	If the Shell is set to be an application shell or top level shell, it is mapped to the Java Frame class. If it is set to be a dialog shell, it is mapped to the Java Dialog class. If, however, the child of the shell is a FileSelectionBox, then both the shell and the FileSelection box are mapped to the one Java FileDialog class.
MessageBox	The MessageBox widget can be mapped to three different Java classes according to the value of the XmNdialogType resource, which can be set in the Settings page of the MessageBox resource panel. The three mappings are: The dialog type is set to Error, Information, Question, Warning or Working. In these cases the widget is mapped to the IconMessagePanel class which is part of the MWT library. The dialog type is set to Message. In this case the widget is mapped to the MessagePanel class which is part of the MWT library. The dialog type is set to Template. In this case the widget is mapped to the Java Panel class combined with the DialogTemplateLayout class from the MWT library.

TABLE 10-1 Mapping of Motif Widgets to Java Components *(Continued)*

Motif Widget	Java Component
MainWindow	The MainWindow widget is, essentially, a composite widget with particular resource settings. To mimic this Sun WorkShop Visual maps the MainWindow to various combinations of Panel and BorderLayout classes according to which resources have been set on the MainWindow and the number of children it has.
DrawingArea	The DrawingArea widget maps to the Java Panel class combined with the DrawingAreaLayout from the MWT library.
DialogTemplate	The DialogTemplate widget maps to the Java Panel class combined with the DialogTemplateLayout class from the MWT.
MenuBar	This maps to the Java MenuBar class.
BulletinBoard	The BulletinBoard widget is mapped internally to the Java Panel class combined with the BulletinBoardLayout class from the MWT library.
Command	This is mapped to the CommandPanel class from the MWT library.
Menu	The Menu widget is mapped to the Java Menu class. Note that this is a Pulldown menu - Popup menus are not mapped to a Java class.
Form	The Form widget maps to the Java Panel class combined with the FormLayoutManager class from the MWT library.
SelectionPrompt	The SelectionPrompt widget is based on the SelectionPanel class from the MWT library. To differentiate this widget from the SelectionBox, which also maps to the SelectionPanel class, internal class methods are called.
RadioBox	This widget maps to the Java Panel class combined with the Java CheckBoxGroup class and a layout manager.
PanedWindow	The PanedWindow widget maps to a Java Panel class combined with the PanedWindowLayout class from the MWT library.
SelectionBox	This maps to the SelectionPanel class from the MWT library.
RowColumn	The RowColumn widget maps to the Java Panel class combined with the ColumnPackedRCLayout class or the TightPackedRCLayout class (both from the MWT library) according to whether the packing resource has been set to “Column” or “Tight” respectively.

TABLE 10-1 Mapping of Motif Widgets to Java Components *(Continued)*

Motif Widget	Java Component
ScrolledWindow	If the scrolling policy resource for the ScrolledWindow is set to "Automatic", the widget is mapped to the ScrolledPanel class from the MWT library. If the scrolling policy is *not* set to "Automatic", the widget is mapped to the ScrollablePanel class, also from the MWT library.
FileSelection	The FileSelection widget maps to the Java FileDialog class. Since the Java class is a dialog, the Shell parent of the FileSelection widget is included in this mapping; both widgets become the one Java class.
Label	If the label type resource is set to "Pixmap", the widget is mapped to the ImageArea class from the MWT library. Otherwise, the Label widget is mapped to the Java Label class.
CascadeButton	This widget is not mapped at all. The label string resource of this widget becomes a resource of its Menu child.
TextField	This is mapped to the Java TextField class.
PushButton	If the label type resource is set to "Pixmap", the widget is mapped to the ImageButton class from the MWT library. Otherwise, the mapping of the PushButton widget is dependent on whether it is in a menu or not. If the PushButton is *not* in a menu, it is mapped to the Java Button class. If it is in a Pulldown menu, it is mapped to the MenuItem class. If the PushButton widget is in an OptionMenu, the label string resource is retained as a resource of the parent Choice class.
OptionMenu	If the Label widget of the OptionMenu has a label string resource set, then the OptionMenu is mapped to the Panel class with one Label and one Choice as its children combined with the FlowLayout class. If, however, the Label child of the OptionMenu has *no* label string set, then the OptionMenu is mapped to the Choice widget. All of these widgets are from the standard Java classes.
Text	The Text widget is mapped to the Java TextArea class.
ToggleButton	If the ToggleButton widget is not in a menu, it is mapped to the Java CheckBox class. If it is in a menu, it is mapped to the Java CheckBoxMenuItem class.
Separator	If the Separator widget is in a menu, it is mapped to the Java MenuItem class with its label set to dash (-). If the Separator is not in a menu, it is mapped to the Separator class from the MWT library.
ScrolledText	The ScrolledText widget is mapped to the Java TextArea class.

TABLE 10-1 Mapping of Motif Widgets to Java Components *(Continued)*

Motif Widget	Java Component
DrawnButton	The DrawnButton widget is mapped to the ImageButton class from the MWT library.
Scale	If the Scale widget has no children it is mapped to the Scale class from the MWT library. If it does have children, it is mapped to the ScalePanel class, also from the MWT library.
List	The List widget is mapped to the Java List class.
ArrowButton	The ArrowButton widget is mapped to the ArrowButton class from the MWT library.
ScrollBar	The ScrollBar widget maps to the Java ScrollBar class.
ScrolledList	The ScrolledList widget maps to the Java List class.

Mapping Motif Widgets to Swing Components

When you generate your Motif design as Java code using Swing components, Sun WorkShop Visual tries to map each Motif widget in your design to a Swing component. If no clear counterpart can be found, Sun WorkShop Visual uses its own MWT library to “mimic” a Swing component and generate code for the widget. The table below illustrates how each Motif widget is mapped.

TABLE 10-2 How Each Motif Widget is Mapped for Swing Generation

Motif Widget	Swing Mapping
Shell	If the Shell is set to be an application shell or top level shell, it is mapped to a JFrame component. If it is set to be a dialog shell, it is mapped to a JDialog component.
MessageBox	The MessageBox widget can be mapped to two different Java classes according to the value of the XmNdialogType resource, which can be set in the Settings page of the MessageBox resource panel. When the dialog type is set to Error, Information, Question, Warning or Working, the widget is mapped to the IconMessagePanel class which is part of the MWT library. When the dialog type is set to Message, the widget is mapped to the MessagePanel class which is part of the MWT library.

TABLE 10-2 How Each Motif Widget is Mapped for Swing Generation *(Continued)*

Motif Widget	Swing Mapping
MainWindow	This maps to the JPanel component.
DrawingArea	The DrawingArea widget maps to a JPanel component.
DialogTemplate	The DialogTemplate widget maps to a JPanel component.
MenuBar	This maps to a JMenuBar component.
BulletinBoard	The BulletinBoard widget is mapped to a JPanel component.
Command	This is mapped to the CommandPanel class from the MWT library.
Menu	The pulldown Menu widget is mapped to a JMenu component. Popup menus are mapped to the JPopupMenu component.
Form	The Form widget maps to a JPanel component.
SelectionPrompt	The SelectionPrompt widget is based on the SelectionPanel class from the MWT library.
RadioBox	This widget maps to a JPanel component.
PanedWindow	The PanedWindow widget maps to a JSplitPane component if it has no more than two children. If it has, then it simply maps to a JPanel.
SelectionBox	This maps to the SelectionPanel class from the MWT library unless the XmNdialogType resource is set to “command” in which case it is mapped to the CommandPanel class from the MWT library.
RowColumn	The RowColumn widget maps to a JPanel component unless its XmNrowColumnType resource is set to: - “menu bar” in which case it becomes a JMenuBar - “menu pulldown” in which case it becomes a JMenu - “menu popup” in which case it becomes a JPopupMenu - “menu option” in which case it becomes a JComboBox.
ScrolledWindow	If the scrolling policy resource for the ScrolledWindow is set to “Automatic”, the widget is mapped to a JScrollPane component. If the scrolling policy is *not* set to “Automatic”, the widget is mapped to the ScrollablePanel class from the MWT library.
FileSelection	The FileSelection widget maps to a JFileChooser component.
Label	The Label widget is mapped to a JLabel component.
CascadeButton	If the parent of this widget is either a menubar or a pulldown menu, this widget is not mapped at all because the parent widget is mapped to a suitable component. If the parent is neither of these, this widget is mapped to a JLabel component.

TABLE 10-2 How Each Motif Widget is Mapped for Swing Generation *(Continued)*

Motif Widget	Swing Mapping
TextField	This is mapped to a JTextField component.
PushButton	If the parent of this widget is a pulldown menu, the widget is mapped to a JMenuItem component. Otherwise it is mapped to a JButton.
OptionMenu	If the Label widget of the OptionMenu has a label string resource set, then the OptionMenu is mapped to a JPanel component. If, however, the Label child of the OptionMenu has *no* label string set, then the OptionMenu is mapped to a JComboBox component.
Text	The Text widget is mapped to a JTextArea component.
ToggleButton	If the ToggleButton widget is the child of a pulldown menu, it is mapped to a JCheckBoxMenuItem component. If it is the child of a radiobox widget or a rowcolumn widget with the radio enabled resource set to true, then it is mapped to a JRadioButton. If none of the above is true, it is mapped to a JCheckBox component.
Separator	If the Separator widget is in a menu, it is mapped to a JMenuItem component. Otherwise, it is mapped to a JSeparator component.
ScrolledText	The ScrolledText widget is mapped to a JScrollPane component with a JTextArea underneath.
DrawnButton	The DrawnButton widget is mapped to a JButton component.
Scale	This is mapped to a JSlider component.
List	The List widget is mapped to a JList component.
ArrowButton	The ArrowButton widget is mapped to the ArrowButton class from the MWT library.
ScrollBar	The ScrollBar widget maps to a JScrollBar component.
ScrolledList	The ScrolledList widget maps to a JScrollPane component with a JList underneath.

CHAPTER **11**

Designing for Microsoft Windows

Introduction

This chapter describes how you can use Sun WorkShop Visual to design applications to run under Microsoft Windows. Sun WorkShop Visual generates MFC (Microsoft Foundation Classes) code for your design which is then directly portable to Microsoft Windows. This chapter covers the following areas:

1. **Generating the Application** - a brief description of the three *flavors* of C++ code generation and the option of generating dialog template files. This starts on page 359.
2. **Starting in Microsoft Windows Mode** - how to run Sun WorkShop Visual so that it checks your design for its suitability for MFC code generation. See page 360.
3. **The Sun WorkShop Visual Window** - a description of the visual differences you can see when running Sun WorkShop Visual for creating Windows designs. See page 361.
4. **Microsoft Windows Compliance** - details of the checking Sun WorkShop Visual does to make sure that your design is suitable for generating as a Windows application, starting on page 363.
5. **Compliance Failure** - what happens, and what to do, when an imported design fails the Windows compliance check. See page 369.
6. **Using Links** - how to use Links in designs for Microsoft Windows. See page 372.

7. Special notes for particular widgets and resource types - what to bear in mind when using the following:

   a. **Manager Widgets and Layout** (page 373).

   b. **Fonts** (page 375).

   c. **Pixmaps, Bitmaps, and Icons** (page 376).

   d. **Colors** (page 377).

   e. **DrawingAreas** (page 381).

8. **Using Third Party Widgets** - how to use third party widgets (or user widgets) in your Microsoft Windows design. See page 377.

9. **Method Declarations** - how to control where methods are declared in the generated MFC code. See page 380.

10. **Application Class** - how to change the generation of CWinApp in the MFC code. See page 383.

11. **File names** - a note on some points you should bear in mind when naming files to be ported to Microsoft Windows. See page 384.

12. **Code Generation** - a description of the code generated for Microsoft Windows, starting on page 385.

13. **Configuring Sun WorkShop Visual** - a description of the resources relevant to creating designs for Microsoft Windows. See page 388.

Chapter 12 ”Creating a Microsoft Windows and Motif Application“ provides step-by-step instructions for creating a Windows-compatible design and generating the MFC code for it. Following this tutorial would give you a detailed insight into the use of Sun WorkShop Visual for creating Windows applications.

Prerequisites

This chapter makes certain assumptions about your knowledge of Sun WorkShop Visual. It is assumed that you understand structured code generation, as described in “Structured Code Generation” on page 249 and that you are comfortable with the idea of generating C++ code. “C++ Classes” on page 254 provides information on C++ in Sun WorkShop Visual.

You may wish to follow the tutorial in Chapter 9 ”C++ Code Tutorial“ to gain experience with the generation of C++ code.

Generating the Application

The best way to develop an application which is to be ported to different platforms is to encapsulate the platform specific parts in some way so that the body of the application is isolated from the implementation details. Sun WorkShop Visual uses its C++ code structuring capabilities to generate a set of classes for the user interface that have the same public interface, but two different implementations. In Sun WorkShop Visual these implementations are known as *flavors*. There are three flavors of C++ code that can be generated:

1. *Motif* – The “vanilla” flavor is the same as generating C++ code when not using the Windows-aware version of Sun WorkShop Visual. The base classes are a very simple set provided, with source, with the Sun WorkShop Visual release.

2. *Microsoft Windows MFC* – The target base classes on the Microsoft Windows platform are the Microsoft Foundation Classes. These provide a fairly high level set of controls and functions which can be used to build user interfaces.

3. *Motif XP* – A set of base classes supplied with source as part of the Sun WorkShop Visual release. They are very similar to the Motif flavor base classes except that they are named to match the Microsoft Foundation Classes and provide a little of the basic functionality. They are not intended to provide the whole of the MFC interface, only enough to allow the developer some measure of shared code in the user interface. The real goal is in sharing code in the rest of the application.

Using Dialog Templates

When you generate code for Microsoft Windows using MFC, you are given the choice of:

1. Generating the dialogs in your design as dialog templates, which are Microsoft Windows resource files describing the dialog.

2. Generating the design in MFC code.

For the first option above, Sun WorkShop Visual generates some MFC code to fetch widgets out of the dialog templates and to control the dialogs, thereby generating a complete application.

- **To generate dialogs as resource files, set the “Generate as Resources” toggle in the Code Options dialog.**

If the “Generate as Resources” toggle is not set, plain MFC is generated for your design. By default, the toggle is set.

The full capabilities of Sun WorkShop Visual's C++ model can still be used for MFC applications. This means that you can use sub-classing and inheritance to add additional functionality. These techniques can be used, for example, to support cross platform versions of your user-defined widgets.

Starting in Microsoft Windows Mode

There are three methods of invoking Sun WorkShop Visual so that it is running in Microsoft Windows mode. Select the method which suits you best, bearing in mind that you may have to share your copy of Sun WorkShop Visual with other users who do not want to run Sun WorkShop Visual in Microsoft Windows mode.

The Resource File

Microsoft Windows mode can be specified by a resource in the file containing your Sun WorkShop Visual application resources:

```
visu.windows:true
```

The Command Line Switch

Microsoft Windows mode can also be specified by a command line switch when invoking Sun WorkShop Visual:

```
visu -windows
```

Separate Version of Sun WorkShop Visual

The third method of invoking Sun WorkShop Visual in Microsoft Windows mode gives the appearance of a separate application which is always in Microsoft Windows mode. This method uses the application resource, described above.

1. **Create a hard link to your Sun WorkShop Visual shell script in the $VISUROOT/bin directory.**
2. **Give this file a name such as `visuwin`.**
3. **Create a hard link to your Sun WorkShop Visual binary in the $VISUROOT/lib directory. Use the same file name as in Step 2.**

4. **Add the windows flag to the** *.Xdefaults* **file in your home directory, using the name of your symbolic link, as follows:**

```
visuwin.windows:true
```

In this way, you can simply type

```
visuwin
```

This will invoke Sun WorkShop Visual in Microsoft Windows mode. Other people can then continue to use Sun WorkShop Visual safe in the knowledge that the default will not be Microsoft Windows mode.

Note – This is the technique that Sun WorkShop Visual uses to bring up the version for smaller screens - such as the VGA screen. The program `small_visu` is nothing more than a hard link to the Sun WorkShop Visual binary which picks up an alternative set of resources from the Sun WorkShop Visual resource file.

The Sun WorkShop Visual Window

The Sun WorkShop Visual Window looks slightly different when in Microsoft Windows mode. The main differences are the addition of two items in the toolbar at the top (and in corresponding menus) and the fact that some of the widgets are not included in the widget palette. This second point is dealt with in “Microsoft Windows Compliance” on page 363. The first set of differences is described here.

Microsoft Windows Compliant Toggles

There are two Microsoft Windows Compliant toggles - one on the toolbar and one in the Module menu. They both have the same function.

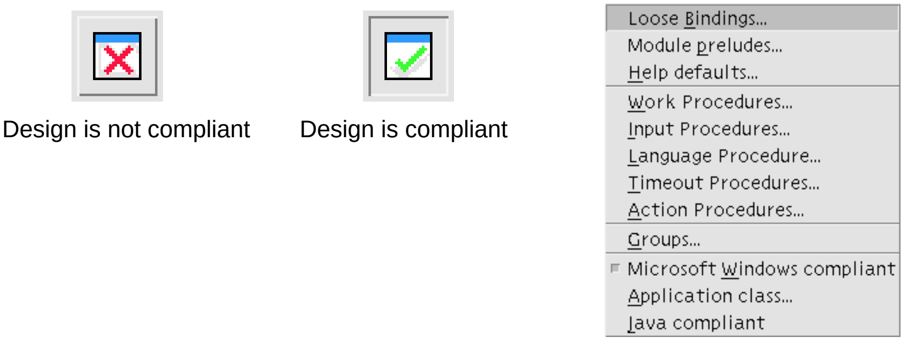


FIGURE 11-1 The Compliance Buttons in the Toolbar (left) and the Module Menu (right)

These toggles indicate whether or not the current design is Microsoft Windows compliant. If you read in a design created with a version of Sun WorkShop Visual not in Microsoft Windows mode or you cut and paste areas of a compliant hierarchy, it is possible to create structures which are not Microsoft Windows compliant.

In such a case a message is displayed informing you where the problem is and the two Microsoft Windows Compliant toggles are unset. The toolbar toggle displays a red cross when it is unset. Having made the appropriate changes to your design, pressing either of these toggles will check the compliance again. If your design is now Microsoft Windows compliant, the toolbar button will be set (the red cross will disappear). If not an error message will appear indicating the problem and the toggles will remain unset.

The Flavor Menu

The flavor menu in the toolbar specifies which flavor of code you wish to generate: plain Motif, Motif XP or Microsoft Windows. This only applies to C++ code generation.

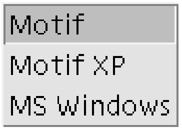


FIGURE 11-2 The Flavor Menu in the Toolbar

Visual Compliance Indicators

In addition to the compliance issues which prevent code from being generated for Microsoft Windows, there are many attributes of a design which have no effect in the Microsoft Windows implementation (for example the alignment of a label on a button), because the Microsoft Windows toolkit does not support the concept. Sun WorkShop Visual indicates this by means of the following:

1. **Icon Cues**. In the resource panels, those resources which have no effect on Microsoft Windows are indicated with a cross icon, those that do have an effect are indicated with a tick icon.
2. **Color Cue**. By default, text input fields and option menus in the resource panels use a pink color to indicate that setting the resource will have no effect in the Microsoft Windows flavor.[1]

If the variable name field of the selected widget is pink[1], this indicates that the widget will not map to any equivalent Microsoft Windows object. For instance, in Microsoft Windows a menubar is simply an attribute of a Dialog, there is no menubar object. Consequently Sun WorkShop Visual will show the variable name pink for menubar widgets.

Sun WorkShop Visual will also use this technique to indicate that some links will not be reproduced in the Microsoft Windows code. Links are discussed in more detail in "Using Links" on page 372.

Microsoft Windows Compliance

Unfortunately for the user interface developer, the Motif and X toolkits bear little resemblance to the Microsoft Windows toolkit. Although the visual appearance is similar, the actual use of the toolkit is very different. This requires Sun WorkShop Visual to impose some restrictions on the developer before Microsoft Windows code can be generated for a design. Sun WorkShop Visual will impose these restrictions when it is in *Microsoft Windows mode*. When Sun WorkShop Visual is in Microsoft Windows mode it needs to permit the developer to work on designs which do not comply with these restrictions (so that an existing design can be read in for example). As a result Sun WorkShop Visual will check that a design is *Microsoft Windows compliant*. If a design is not Microsoft Windows compliant then C++ code can only be generated for the Motif flavor.

[1] Pinking is only discernible on a color display.

This section details the restrictions imposed by Sun WorkShop Visual which ensure that Microsoft Windows compliant code can be generated. When in Microsoft Windows mode Sun WorkShop Visual will not allow you to create a design which violates these restrictions.

Structure Restrictions

Because of the differences between Motif and Microsoft Windows in the way events are handled, some widgets cannot be made classes. In Microsoft Windows, events concerning certain widgets are always sent to the enclosing class. Other widgets must be classes in order to handle Microsoft Windows messages. Here is a list of these restrictions:

- Cannot be class
  - MenuBar
  - PopupMenu
  - CascadeButton
  - OptionMenu
  - Any widget which is the child of a Shell
- Must be class
  - Shells
  - ScrolledWindow (unless child of Shell)
  - Frame
  - RadioBox, unless the child of a Frame
  - DrawingArea (unless child of MainWindow, ScrolledWindow or Shell)
  - Paned Window
  - Child of Paned Window

The first error you are likely to encounter on reading a non-compliant design is the fact that the Shell must be structured as a C++ class. This error is easy to fix and can be done automatically from the Compliance Failure dialog. See “Compliance Failure” on page 369 for more details.

C Structures

Sun WorkShop Visual does not support the Function or Data Structure options in a compliant design.

Classes and Callbacks

Structural errors can be considerably more complicated if you have a design which is well-structured, making good use of C++ classes and with callback methods scattered among the child widgets. The following example demonstrates how this problem may occur and how to overcome it.

Example

When a widget is given a callback method, the method is declared in the enclosing class. In the following example, while using a version of Sun WorkShop Visual which was not in Microsoft Windows mode, the MenuBar, MBar_class, was declared a class and the button given a callback method:

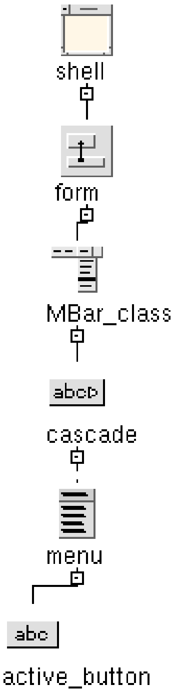
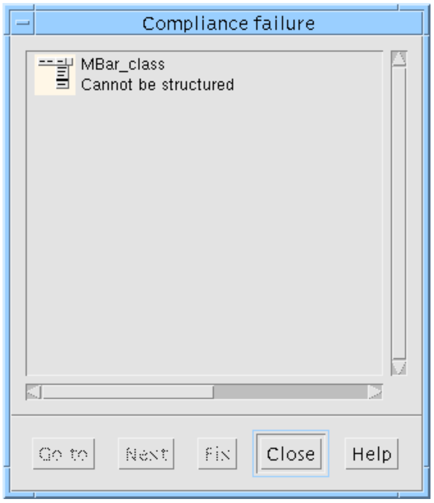


FIGURE 11-3 Non-Compliant Hierarchy

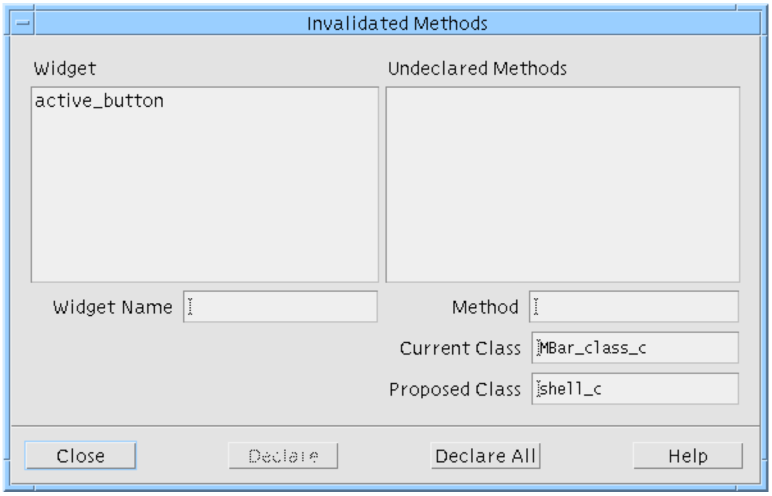
The method is declared in MBar_class. If you then read the design into a version of Sun WorkShop Visual in Microsoft Windows mode, you will be presented with the following error message because MenuBars cannot be classes:



MBar_class
Cannot be structured

FIGURE 11-4 Error Message Showing Non-Compliance

If you remove the class definition of the MenuBar (using the Core Resources dialog), you will then see the following error message:



active_button

Widget Name

Method

Current Class MBar_class_c

Proposed Class shell_c

FIGURE 11-5 Message Informing of Method Callback Declaration Invalidation

Make sure that the method declarations are removed from the MenuBar and, in this case, added to the Shell by either pressing the “Declare all” button or selecting “active_button” in the list and pressing “Declare”. You cannot add them to the Form because the Form, like the MenuBar, does not map to an object on Microsoft Windows.

Although Sun WorkShop Visual assists you in the above way, changing whether a widget is a class or not could have a major impact on your application. You will need to reconsider the structure of your application very carefully.

Menubar Restrictions

Menubars in Microsoft Windows are created by setting an attribute of the Dialog. This leads to two compliance restrictions:

- Only one Menubar per shell is supported. You cannot have a design which contains more than one Menubar in a Dialog
- The Menubar parent must be the child of the Shell. A Menubar cannot be an immediate child of a Shell, nor can it be at a deeper level in the widget hierarchy than as a child of the Shell’s child

FileSelectionBox

The File Selection Box must be a child of a DialogShell or TopLevelShell. In Microsoft Windows file selection is provided by a pre-defined FileSelection Dialog. This dialog can only contain a single work area child, it cannot support a Menubar, nor additional buttons (not managed by the work area).

Unsupported Widgets

The following widgets have no comparable control in Microsoft Windows and so cannot be used in a design that is to be portable:

- SelectionBox
- Command
- MessageBox
- SelectionPrompt
- DrawnButton
- ArrowButton

All but the last two buttons, however, can be emulated using the Dialog Template widget, which is supported as a Windows control.

Scale

The Scale widget maps to a Microsoft Windows ScrollBar control which cannot support child controls. Therefore a Scale widget with children violates the Microsoft Windows compliance restrictions.

Frame and RadioBox

Because of the way messages are passed to an enclosing control, both Frame and RadioBox (if not the child of a Frame) have to be classes. However, as the child of a Shell cannot be a class, it follows that neither Frame nor RadioBox can be the immediate child of a Shell.

The second child of a Frame must be a Label widget - this is the title of the Frame. The Frame control in Microsoft Windows (actually a CButton in disguise) simply has a title attribute. There is no way to use another control as the title. The first child can be any widget.

MainWindow and ScrolledWindow

Microsoft Windows does not support automatic scrolling and hence Sun WorkShop Visual in Microsoft Windows mode disables automatic scrolling options. The MainWindow widget may only include a work area and Menubar child. It does not support the command window or message window.

Paned Windows

A compliant design cannot contain a Paned Window which has Separator, MainWindow, OptionMenu, or MenuBar children. Neither may the children be definitions or instances.

Definitions

The XmNlabelType resource cannot be explicitly set for a widget which is a component of a definition when it is instantiated in another design. If a Button does not have XmNlabelType set in a definition then when that definition is used XmNlabelType cannot be set to PIXMAP. This is because Microsoft Windows uses a different class (CBitmapButton instead of CButton) to implement a button with a bitmap on it. It is obviously not possible to change the class of a variable after it has been created, hence the restriction.

For the same reason it is not possible to have a CascadeButton in a definition which has no Pulldown menu and to then add the Pulldown in an instance.

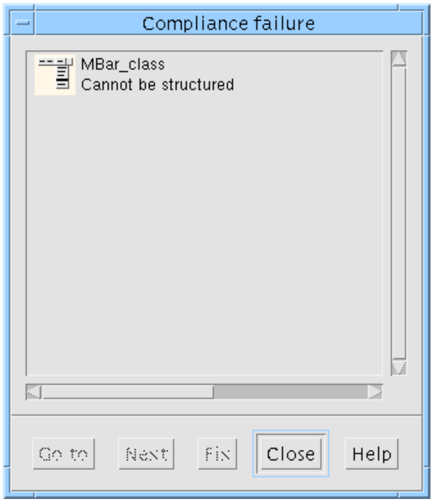
You cannot make the child of a shell widget the root of a definition because you cannot make the child of a shell a C++ class, as explained in “Structure Restrictions” on page 364. To overcome this restriction, simply add a “dummy” container between the shell and the widget you wish to use as the root of your definition.

Slightly more subtly, it is not possible to have a widget with methods added to an instance which is not being sub-classed. For example, if you have an instance of a RowColumn definition, and the root widget (i.e. the RowColumn) does not have its structure set to class. When not in Microsoft Windows mode it is possible to add a button to the RowColumn instance and give it a method callback which is declared in an enclosing scope (say a Shell class). This is not possible in Microsoft Windows; the event has to be handled by the enclosing control (in this case the CWnd which represents the RowColumn).

The widgets in a definition must be named. This is a requirement for all C++ definitions, not just those destined for use under Microsoft Windows.

Compliance Failure

When you read in a design which was created by Sun WorkShop Visual while not in Microsoft Windows mode or you use cut and paste in such a way as to cause a design to become non-compliant, the Compliance Failure dialog appears showing you which widgets are causing the design to be non-compliant.



MBar_class
Cannot be structured

FIGURE 11-6 The Compliance Failure Dialog

You can select the widgets in the scrolled list of non-compliant widgets. You may then press one of the following buttons:

- *Go to* – This highlights the widget in your design. If the widget is part of a dialog which is not the current one, the relevant dialog is selected. If the widget is in a part of the design which is folded, the design is unfolded. *Double-clicking* on a widget is the same as pressing *Go to*. For structure problems, Sun WorkShop Visual will also open the Core resources dialog on the Code Generation page.
- *Next* – This moves to the next widget in the list and selects the widget in the hierarchy.
- *Fix* – This fixes the problem, where possible. Sun WorkShop Visual can only fix those errors connected with the structure of a widget. Other errors cannot be fixed automatically. See "Microsoft Windows Compliance" on page 363 for a list of the possible compliance errors.

Example

This example shows how compliance failure is detected. In the hierarchy shown in Figure 11-7, the RowColumn widget *rc_is_class* has been made a class:

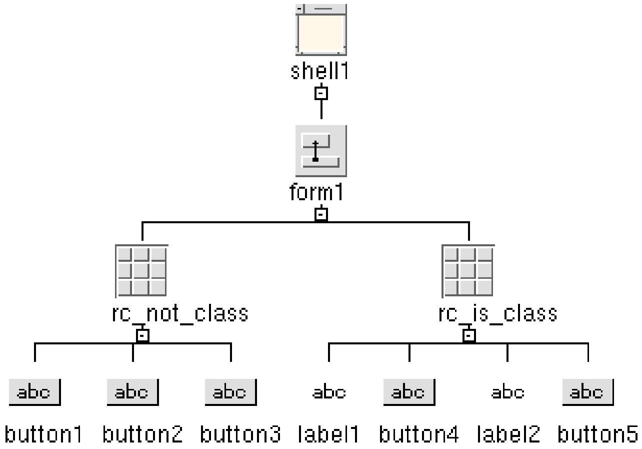
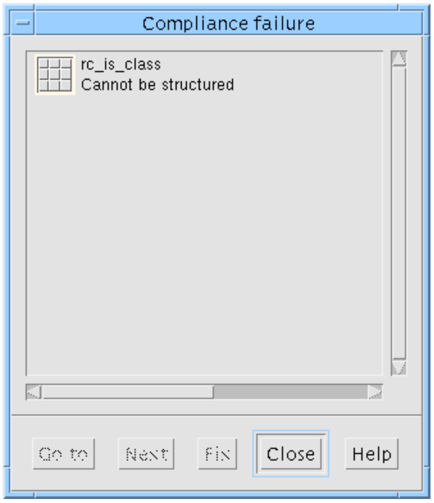


FIGURE 11-7 RowColumn Defined as Class

If you were to cut *rc_is_class*, clear the form and then paste (i.e. pasting *rc_is_class* as a child of shell), Sun WorkShop Visual would mark the design as non-compliant and display the Compliance Failure Dialog shown in Figure 11-8 indicating that the child of a shell may not be a class:



rc_is_class
Cannot be structured

FIGURE 11-8 Error When Pasting a Class as Child of Shell

Sun WorkShop Visual allows you to continue with the operation but until you change the structure of *rc_is_class*, you will have a design which is not Microsoft Windows compliant. The Microsoft Windows compliant toggle in the toolbar displays a red cross to remind you.

Using Links

In the Motif flavors links are pre-defined callbacks. In the Microsoft Windows flavor they are implemented as simple global functions which are called by a button’s message handler. There are, however, some restrictions on how links can be used on Microsoft Windows. These restrictions only affect whether code is generated for a link, they do not affect the design’s compliance.

Destination Widget Not an Object on Microsoft Windows

If the widget selected as the destination of a link is not mapped to an object on Microsoft Windows, the Add button in the Edit Links dialog is pink. The link can still be added and will be effective on Motif but it will not be generated into the Microsoft Windows code. To indicate this, the widget in the list of links is pink. See “Mapping Motif Widgets to Microsoft Windows” on page 782 for more information on which widgets are mapped to Microsoft Windows objects.

Buttons in Menus as Link Destinations

When adding a link where the destination widget is a PushButton in a Menu, the type of link is restricted to enable and disable. You cannot show, hide, manage or unmanage a Menu Button on Microsoft Windows.

File Selection Dialog as Link Destination

FileSelectionBox is implemented on Microsoft Windows as a CFileDialog, a modal dialog which is shown by calling the DoModal method. This method does not return until the FileSelection is complete or cancelled. On Microsoft Windows, therefore, only the show link is supported. For both MFC and Motif XP flavors the code is structured so that the DoModal method does not return until the selection is complete or cancelled.

Manager Widgets and Layout

The Motif manager widgets have no equivalents on Microsoft Windows. Widgets such as Forms and RowColumns do not exist on Microsoft Windows. If you have one of these in your design which is not a class, it is ignored in the generated code. If it is a class, Sun WorkShop Visual generates a Canvas in its place.

If you are generating dialog templates from your design, most manager widgets are ignored because dialog templates tend to be *flat*. They do not use the containment hierarchy that is common on Motif. Your design, therefore, is flattened out in the template files. Widgets are taken out of sub-containers and made children of the dialog. This also enables system resources to be used throughout the design.

If you choose to generate your design into Windows resource files (or dialog templates) along with the main application code, you will be able to make adjustments to the layout using your Microsoft Windows IDE (Integrated Development Environment). These changes cannot be taken back into Sun WorkShop Visual but they do allow you to make quick, simple adjustments right where they are needed.

If you do not wish to generate dialog templates, Sun WorkShop Visual generates code containing absolute values for sizes and positions as they are at the moment of generation. So, for example, if you have a PushButton that is 100 pixels wide, 30 pixels high and is located at 10, 200 then those explicit values will be used in the Microsoft Windows code, even though you have not explicitly set the x, y, width and height resources but allowed them to be calculated by the Motif toolkit. In practice this gives very good results - generating Microsoft Windows dialogs which look very similar to their Motif counterparts.

Fonts and Appearance

Because Sun WorkShop Visual is generating an absolute size for a Microsoft Windows control, it is important that the size of a dialog is appropriate for any font that will be used for text displayed in it. The best way to ensure this is to make Sun WorkShop Visual use a similarly sized font to display the dialog while it is being designed. There are two ways to do this.

The first way is to force the control to use a particular sized font, perhaps by setting the XmNfontList resource for the control or by setting the appropriate font resource on the shell. Consequently the dialogs will be similarly sized.

Alternatively use a resource so that Sun WorkShop Visual displays the design windows with a font that approximates to the default font used on Microsoft Windows. This will cause Sun WorkShop Visual to generate absolute sizes that are appropriate to the font. The Motif code will use a default font in the normal way. To make Sun WorkShop Visual use a specific font for the design on Microsoft Windows, use settings similar to the following in your resource file:

```
visu*dialog.labelFontList:\
-adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
visu*dialog.buttonFontList:\
-adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
visu*dialog.textFontList:\
-adobe-helvetica-medium-r-normal-*-14-*-*-*-*-77-iso8859-1
```

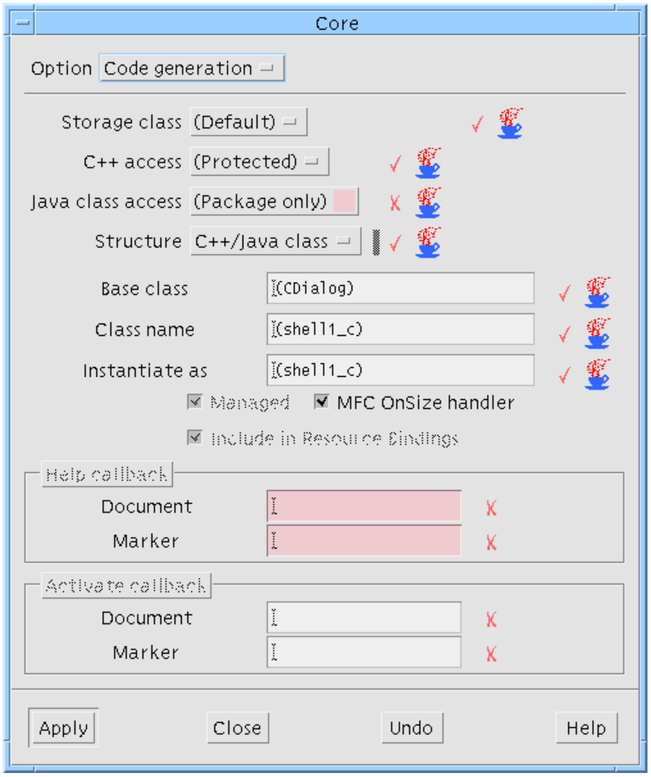
These values may work well for you, but it will depend on the precise font used on your Microsoft Windows system. It is the size and average width values which are important.

Resize Behavior

In Microsoft Windows mode, Sun WorkShop Visual generates OnSize message handlers to provide some resize behavior when the user resizes a dialog. Sun WorkShop Visual does not attempt to reproduce exactly the Motif geometry management, rather it generates a handler which will simulate the resize behavior of certain manager widgets. In particular, these are:

- ScrolledWindow
- Form
- Frame
- DialogTemplate

These managers do not need to be classes in order to produce the resize behavior; Sun WorkShop Visual generates a handler for the enclosing class that handles all descendant widgets. You can suppress the generation of the resize handler if you want to provide your own (through a sub class for example), by unsetting the MFC OnSize handler toggle on the Code generation page of the Core resources dialog. See Figure 11-9.



Core

Option: Code generation

Storage class: (Default) ✓

C++ access: (Protected) ✓

Java class access: (Package only) ✗

Structure: C++/Java class ✓

Base class: (CDialog) ✓

Class name: (shell1_c) ✓

Instantiate as: (shell1_c) ✓

☑ Managed ☑ MFC OnSize handler

☑ Include in Resource Bindings

Help callback

Document: ✗

Marker: ✗

Activate callback

Document: ✗

Marker: ✗

FIGURE 11-9 The Core Resources Dialog - Code Generation Page

Fonts

In order for a font to be generated into the Microsoft Windows code *you must use font objects*. This is because fonts must be persistent on Microsoft Windows. Only by using font objects can you guarantee this. Sun WorkShop Visual provides a visual cue by making the Apply button pink if you select an item from the list of fonts in the font selection dialog. If, however, you select an item from the list of font objects, the Apply button is no longer pink.

Fontlists and Compound Strings

If you specify fontlists for your objects in Sun WorkShop Visual, the first font in the list will be used for the object on Microsoft Windows. Compound strings containing a mixture of tags will be translated to Microsoft Windows using only the first font specified.

Font Naming

Fonts in Microsoft Windows are named using a different mechanism from that used by X Windows. However, Microsoft Windows has quite a sophisticated matching algorithm. So, although Sun WorkShop Visual uses a fairly crude mapping to translate the font specification, even if you specify a font which is not available in Microsoft Windows, it will probably be substituted with something that looks reasonable.

Pixmaps, Bitmaps, and Icons

Pixmap objects created in Sun WorkShop Visual are converted into a Microsoft Windows bitmap or icon (depending on whether the object is a button or label respectively). This is done for you automatically when you generate a Microsoft Windows resource file. X monochrome bitmaps are not supported in the translation to Microsoft Windows.

When you select “Microsoft Windows Resources” from the Generate dialog, Sun WorkShop Visual informs you that it will create a resource file and a bitmap/icon file for each pixmap you have created. Bitmap files are generated with the suffix “.bmp” and icon files with the suffix “.ico”. You never need to save the pixmap to a file for Microsoft Windows but you may wish to do so for the Motif version. Note that icons are always scaled to 32x32 pixels on Windows.

Buttons With Pixmaps

For a Motif Button with a pixmap type label, Sun WorkShop Visual generates a CBitmapButton for Microsoft Windows. One difference between buttons with pixmaps on Motif and CBitmapButtons on Microsoft Windows is that CBitmapButtons have no border - in fact they do not look like buttons at all. You may, therefore, wish to incorporate a border in your pixmap design.

Colors

You can set the Background and Foreground color of a widget which will be mapped to an object on Microsoft Windows. These colors will be generated into the Microsoft Windows code in terms of their RGB (Red, Green, Blue) values. Microsoft Windows does not normally have the richness of color that is commonly available on X/Motif. For this reason the colors may not look identical on the two platforms. By default, however, the colors for the Microsoft Windows 95 “look and Feel” are used.

Color Objects

Specify the Background and Foreground colors in the usual way in Sun WorkShop Visual. Background colors must be color objects - Foreground colors do not have to be.

Using Third Party Widgets

Sun WorkShop Visual can be extended to support widgets from any other Xt toolkit in addition to the default Motif set. Widgets added to Sun WorkShop Visual are called *user-defined widgets or third party widgets*. Chapter 23 ”User-Defined Widgets“ provides a detailed description of this topic. The new widgets appear in the Sun WorkShop Visual widget palette and can be used in the same way as the pre-defined Motif ones.

Although, in Microsoft Windows mode, Sun WorkShop Visual does not provide explicit support for third party widgets, Sun WorkShop Visual’s C++ model allows you some flexibility in this area.

Sun WorkShop Visual’s Automatic Behavior

Sun WorkShop Visual automatically treats third party widgets as though they are an instance of the class from which they are derived. So, for example, if you have a third party widget derived from XmPushButton, Sun WorkShop Visual will add it to your design as though it is an XmPushButton. Sun WorkShop Visual will then generate Motif XP code which is based on the CButton class, since this is the way

XmPushButton is implemented in the Motif XP, to create an instance of your third party widget. The Microsoft Windows code will be exactly the same as if you had used a PushButton.

The example above uses the XmPushButton class which is a class supported by the Motif XP. If, however, the third party widget is derived from a class which is *not* supported by the Motif XP then Sun WorkShop Visual cannot handle that widget and will not generate code for it. In this case, you should use the strategy outlined in the following section.

Configuring Sun WorkShop Visual for Third Party Widgets

If Sun WorkShop Visual's default behavior for a particular third party widget (described above) is not satisfactory, you can use the flexibility of the C++ model to enhance Sun WorkShop Visual's behavior. There are two steps:

1. Decide on a sensible mapping from your third party widget to a Windows control - whether that is a builtin MFC class or another third party component. For example, many third party widget sets contain a ComboBox widget; this would clearly map through to the builtin MFC CComboBox class.
2. Tell Sun WorkShop Visual the name of the class it should use. On the Code Generation page of the Core Resources panel, specify the C++ class structure to be generated for the selected third party widget. Sun WorkShop Visual fills this page in with its own mappings for the Motif components, but does not always know what to do with third party widgets, simply generating a basic Canvas in its place. All you would need to do here is to override the suggested classing with the mapping you feel is appropriate, and Sun WorkShop Visual will generate code accordingly. This means you will get an object of the right type rendered into your MFC code.

On the Unix side, you would need to include a dummy class so that the code compiles there also for either pure Motif C++ or Motif XP, depending on what you want. The following is an illustration of the code you would have to add:

```
/* Pure Motif C++ Code */
#ifdef    XD_MOTIF
#define MY_BASE_CLASS xd_XtWidget_c  /* Use the Base Sun WorkShop
Visual
                                       Widget Class */
#endif /* XD_MOTIF */

/* Motif XP C++ Code (Unix C++ with strict MFC API) */
```

```
#ifdef    XD_MOTIF_MFC
#define MY_BASE_CLASS CWnd /* Use the Base Motif XP “MFC” Control
                            Class */
#endif /* XD_MOTIF_MFC */

/* Windows C++ Code (Real MFC) */
#ifdef    XD_WINDOWS_MFC
#define MY_BASE_CLASS   CComboBox /* Use the Real Mapping MFC
                                Control Class */
#endif /* XD_WINDOWS_MFC */

#if   defined(XD_MOTIF_MFC) || defined(XD_MOTIF)
class MyMappingClass_c : public MY_BASE_CLASS
{
}
#endif /* XD_MOTIF_MFC || XD_MOTIF */
```

For generating Motif XP, you could also extend the Motif XP library. See “Enhancing the Motif XP” on page 835 for details on how to do this

Third Party Widget Resources

With third party widgets which it does not understand, Sun WorkShop Visual has no way of knowing which Motif resources map to which MFC resources. For these widgets you will probably have to add the code for their resources by hand. Here are some possible strategies:

1. Put all the third party widget resources into Loose Bindings so that they go into the X resource file and are maintained inside Sun WorkShop Visual, thus working well on the X side. Then hand-edit the MFC resource file for similar purposes. See “Loose Bindings” on page 86 for more information on how to create and use Loose Bindings.
2. Generate code on Unix with resources and the code options appropriately set, then generate no resources to code for the third party widgets and again maintain the MFC resource file by hand.
3. Specify a code prelude to add the resources you need. This can have anything you like in it, so you could do something like the following as a pre-manage prelude (assuming my_text is a class):

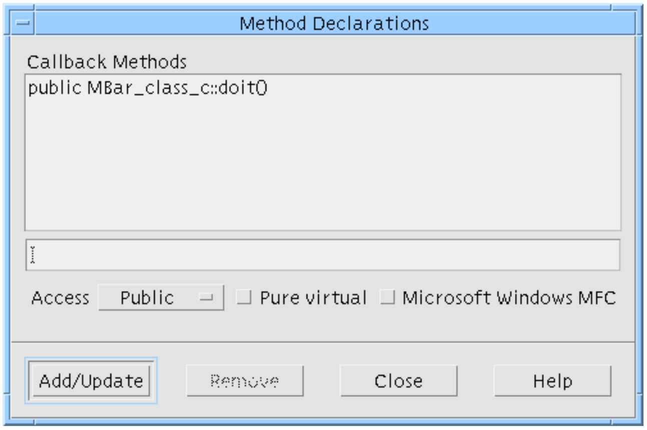
```
#ifdef    XD_MOTIF
XtVaSetValues(my_text->xd_rootwidget(), XmNvalue, “Hello”, 0) ;
```

```
#endif /* XD_MOTIF */

#if   defined(XD_MOTIF_MFC) || defined(XD_WINDOWS_MFC)
my_text->SetWindowText(“Hello World”) ;
#endif /* XD_MOTIF_MFC || XD_WINDOWS_MFC */
```

Method Declarations

The Method Declarations dialog has an extra feature when in Microsoft Windows mode. There is a toggle button labelled Microsoft Windows MFC.



public MBar_class_c::doit()

Access Public ☐ Pure virtual ☐ Microsoft Windows MFC

FIGURE 11-10 Method Declarations Dialog with Microsoft Windows MFC Toggle

This toggle is used to denote whether the method is declared in the class structure of the enclosing class when generating Microsoft Windows code. When generating Motif code, the method is still declared in the enclosing class. The enclosing class is the nearest ancestor which has its structure set to class (either explicitly or automatically).

This toggle does not indicate whether or not the method appears in the Microsoft Windows code stubs - the Callbacks dialog indicates this. If, in that dialog, the method callback has an asterisk (*) appended to it, it will not be generated into the Microsoft Windows stubs file. The Microsoft Windows MFC toggle gives you control over the method declarations, although you would usually use the default that Sun WorkShop Visual provides. Use the Method declarations dialog to declare methods in a class of your choosing or declare them in one of your own classes. They must be declared somewhere.

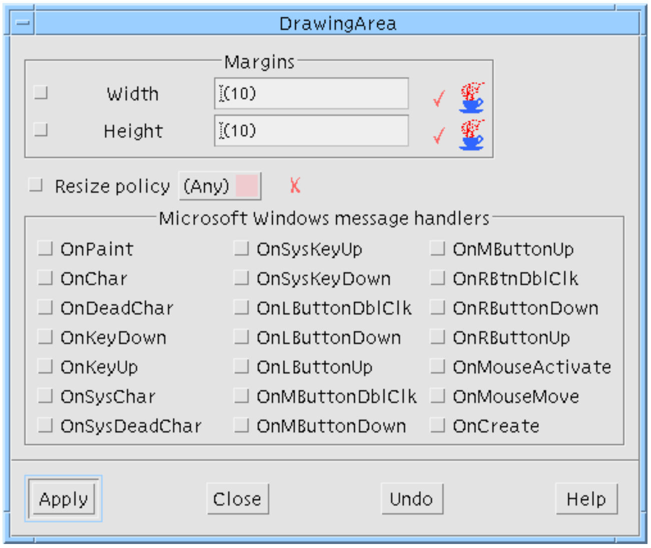
When you add a method callback in the Callbacks dialog, if the method has not been declared already, Sun WorkShop Visual will declare it for you in the enclosing class. If you do not wish the method to be declared there, use the Method Declarations dialog to unset the Microsoft Windows MFC toggle.

DrawingAreas

If a DrawingArea is not the child of a ScrolledWindow, MainWindow or Shell it is created as a basic CWnd class - otherwise it is ignored for Microsoft Windows code generation. See “Mapping Motif Widgets to Microsoft Windows” on page 782 for more information.

Adding Drawing Callbacks for Microsoft Windows

The Motif XP class library does not include any drawing support; any you require will be platform specific. However, Sun WorkShop Visual does allow you to add callback methods which by default are only declared for Motif flavors and Microsoft Windows message handlers. When in Microsoft Windows mode Sun WorkShop Visual adds a set of additional toggle buttons to the DrawingArea resource panel which can be used to add a Microsoft Windows message handler in the generated code.



DrawingArea

Margins

□ Width (10)

□ Height (10)

□ Resize policy (Any)

FIGURE 11-11 The DrawingArea Resource Panel

The Microsoft Windows Message Handler for DrawingArea

If, for example, you were to select the OnRButtonDown toggle in the panel pictured above, the following stub is added to your callback stubs file:

```
afx_msg void scrolled_win_c::OnRButtonDown( UINT nFlags, CPoint point
)
{
}
```

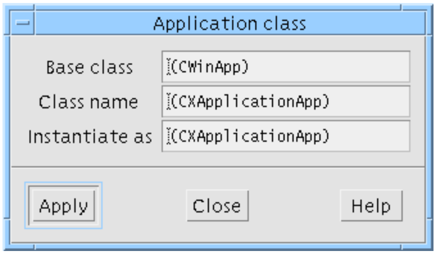
Note that `afx_msg` is a pseudo keyword on Microsoft Windows. The following lines are added to your C++ externs file:

```
//{{AFX_MSG(scrolled_win_c)
afx_msg void OnRButtonDown( UINT nFlags, CPoint point ):
//}}AFX_MSG
DECLARE_MESSAGE_MAP ()
```

This registers the message handler with Microsoft Windows.

Application Class

Sun WorkShop Visual generates an instance of a CWinApp class to the MFC C++ flavors to represent the application. You can configure this class by means of the Application Class dialog which is displayed from the Module Menu. Figure 11-12 shows the dialog.



Base class (CWinApp)

Class name (CXApplicationApp)

Instantiate as (CXApplicationApp)

FIGURE 11-12 The Application Class Dialog

Note – The declaration of the application class (whether defined by yourself or using Sun WorkShop Visual's default) is only generated when you are generating a main procedure - the "Main Program" in the Generate dialog.

Event Handlers

"Event Handlers" on page 203 explains how Sun WorkShop Visual allows you to add event handlers to widgets in your design. Most of the event masks available for Motif widgets can be mapped to Windows code. The following table shows this mapping.

TABLE 11-1 How Event Masks Are Generated into Windows Code

Motif Event Mask	Windows Code
MouseMotion	Generates a generic handler for any mouse movement with or without a button press.
ButtonPress	Generate all three handlers - Left, Center and Right pressed handlers.
ButtonRelease	Generates all three handlers - Left, Center and Right release handlers.
EnterWindow	OnMouseActivate
ExposureMask	OnEraseBkgnd
KeyPressMask	WM_KEYDOWN
KeyRelease	WM_KEYUP
KeymapstateMask	WM_SYSKEYUP / WM_SYSKEYDOWN
LeaveWindowMask	WM_KILL_FOCUS
ResizeRedirect	WM_SIZE
PropertyChangeMask	ON_WM_PAINT
VisibilityChangeMask	WM_SHOWWINDOW

File Names

Filenames on the PC are restricted to 12 characters in total (including the dot) which must be distributed as no more than eight before the dot and no more than three after. If you wish to share files between the two platforms, this restriction will have to be kept in mind when generating Microsoft Windows code. Remember, also, that MS-DOS and Microsoft Windows are not case sensitive. Do not rely on upper and lower case letters to distinguish filenames.

Pixmaps

The above restrictions should also be remembered when naming pixmap objects. When you ask Sun WorkShop Visual to generate a Microsoft Windows resource file after you have created pixmaps, Sun WorkShop Visual automatically generates Microsoft Windows bitmaps and icons in separate files using the name you specified in the pixmap editor. If, therefore, you have specified a name with more than eight characters, you will encounter problems on Microsoft Windows.

C++ Code

Different compilers have varied conventions acceptable on filename extensions. The suffix '.cxx' seems to be universally supported; most compilers should support '.cpp'; some compilation systems may accept '.C' and '.c++'. Visual C++ will complain if you specify '.c' for a file which contains C++ code.

See "Setting the Filename Filter" on page 389 for details of how to change the default filters in the Generate dialog.

Makefile

No makefile can be generated for the Microsoft Windows code files. Since `make` is a UNIX utility, it is assumed that you will be using a Microsoft Windows development environment (such as Visual C++) to build your application.

Note – If you are using Make on your UNIX platform, it is best to generate different code flavors into separate directories in order to avoid confusion and name clashes.

Code Generation

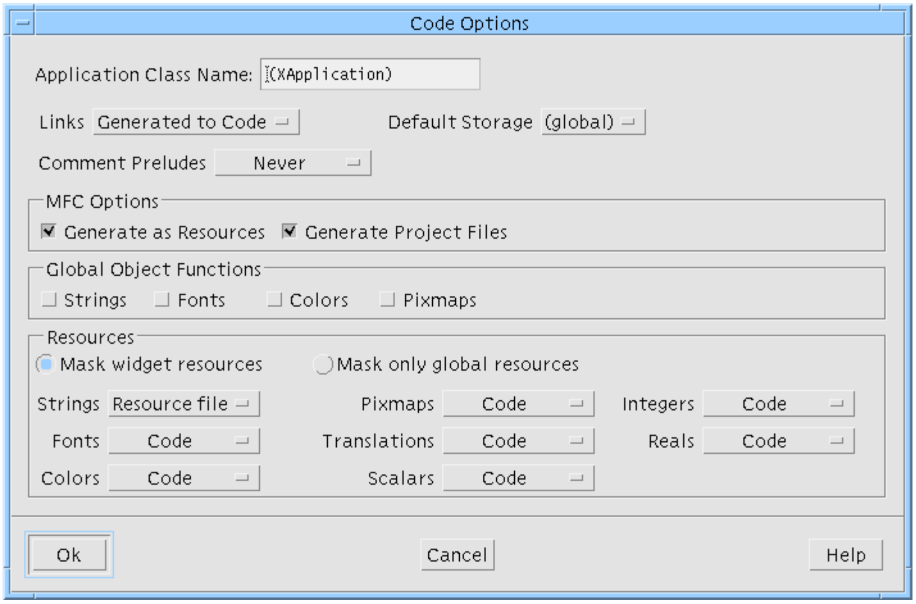
There are a number of points connected with code generation which you should understand before generating code. These are detailed in the following subsections. Briefly, they are:

1. Generating Dialog Templates. How you can generate your dialogs as Microsoft Windows resource files which are more easily manipulated in your Windows environment.

2. Project Files. The Visual C++ project files that Sun WorkShop Visual generates for you.
3. Synchronizing Save and Code Files. How to make sure that your save file and generated code file remain in synch.
4. Dialog Flashing. A warning that dialogs are realized (and therefore displayed) when MFC code is generated.
5. Use of Japanese Font. What to do when generating MFC code containing Japanese text.

Generating Dialog Templates

When you generate code for MFC, the Generate Options dialog (displayed by pressing the “Options” button at the bottom of the Generate dialog) contains a toggle labelled “Generate as Resources”. This is shown in Figure 11-13. When this toggle is set, Sun WorkShop Visual generates the dialogs in your design as dialog templates, which are Microsoft Windows resource files describing the dialog. Sun WorkShop Visual generates some MFC code to fetch widgets out of the dialog templates and to control the dialogs, thereby generating a complete application. If the “Generate as Resources” toggle is not set, plain MFC is generated for your design. By default, the toggle is set.



Code Options

Application Class Name: (XApplication)

Links: Generated to Code   Default Storage: (global)

Comment Preludes: Never

MFC Options

☒ Generate as Resources   ☒ Generate Project Files

Global Object Functions

□ Strings   □ Fonts   □ Colors   □ Pixmaps

Resources

● Mask widget resources   ○ Mask only global resources

Strings	Resource file	Pixmaps	Code	Integers	Code
Fonts	Code	Translations	Code	Reals	Code
Colors	Code	Scalars	Code		

FIGURE 11-13 Generate Options Dialog

Using dialog templates gives you the following advantages:

1. You can use your system default resources - such as font settings. This is not possible with plain MFC.

2. You can use your Microsoft Windows IDE to alter the layout of dialogs easily. Note, however, that you cannot take changes back to Sun WorkShop Visual.

Designs for which resource files (or dialog templates) are generated are dialog based. They do not have a main Frame window. This gives you a much closer correlation between what you see on Motif and what you see on Microsoft Windows.

The Application Shell

In the generated code, an Application Shell inherits from a CDialog. This means that pressing <Enter> when the dialog itself is selected closes the application. As this is the case two of the dialog's methods are generated to the stubs file to be overridden if this behavior is not desired. The methods are OnOK and OnCancel. As with all methods in the stubs file, anything added to them is retained when code is regenerated.

Project Files

When you generate any type of MFC code for Microsoft Windows, you can also generate project files for use in Microsoft Visual C++ using the base name specified in the Generate dialog. To make Sun WorkShop Visual generate project files, check that the "Generate Project Files" toggle is set in the Code Options dialog.

The files generated are:

- `<generate_filename>.dsw` - This is the main project file. It uses the information in the following file.
- `<generate_filename>.dsp` - this contains information about the generated files.

To use these, open the main project file (with the suffix "dsw") in Microsoft Visual C++. These project files are suitable for Visual C++ version 5 and version 6.

Synchronizing Save and Code Files

Sun WorkShop Visual has to store the widget id numbers in the save file for definitions so that code can be correctly generated for instances which indirectly modify the layout of an unnamed component. This can cause a problem if the design is changed in a way which affects the widget ids (such as resetting) before the code is generated. Sun WorkShop Visual will detect such loss of synchronization and will prompt you to save the file.

Dialog Flashing

In order for Sun WorkShop Visual to correctly generate layout information the dialogs need to be realized. Sun WorkShop Visual will automatically show and then hide any unrealized dialogs when Microsoft Windows code is generated. You may see the dialogs appear briefly on the display.

Use of Japanese Font

Any Microsoft Foundation Class (MFC) code generated by Sun WorkShop Visual which contains Japanese text needs to be post-processed before it is compiled under Microsoft Windows.

A filter utility, xdtosj, is provided as part of the Sun WorkShop Visual release to perform this conversion. xdtosj converts the MFC code from EUC to Shift-JIS encoding and changes the DEFAULT_CHARSET value in the MFC CFont creation method to SHIFTJIS_CHARSET.

xdtosj is used in the following way:

```
xdtosj [-xf*] [file]
```

The arguments are:

`-x` Displays a brief explanation of the utility

`-f filter` The shift-jis filter to use (default jconv)

`-*` Any other ‘-’ flags are passed to the filter

`[file]` An optional file. If no file is specified, stdin is used

Configuring Sun WorkShop Visual

There are a number of application resources which apply to Sun WorkShop Visual in Microsoft Windows mode. One of these is the windows flag, indicating that Sun WorkShop Visual should start up in Microsoft Windows mode. This is described in “Starting in Microsoft Windows Mode” on page 360.

Adding Ctrl-M to Generated Lines

By default, Sun WorkShop Visual generates code for Microsoft Windows which adds *Ctrl-M* as the carriage return before the linefeed character. This character is expected by MS-DOS and may, therefore, be expected by your file transfer program. If, however, you do not wish to have these characters at the end of each line, make the following change to the Sun WorkShop Visual resource file:

```
visu.mfcCarriageReturn:false
```

Setting the Color of Non-Microsoft Windows Resource Fields

By default, Sun WorkShop Visual indicates that a field in a resource panel, or a button or any other text field, is not applicable to Microsoft Windows by coloring it pink. This color can be changed by altering the following line in the Sun WorkShop Visual application resource file:

```
visu.mfcTextWarningBackground:#ecc9c9eacdda
```

The example above shows the default file entry - i.e. the color pink. You can change this large number to a more readable color name.

Setting the Filename Filter

In the Generate dialog, Sun WorkShop Visual provides a default filename filter in the Filter text field. You can change this in the application resource file. Search for the following:

```
visu.c++Filter:*.c
visu.c++StubsFilter:*.c
```

These are the default entries for Motif code generation - both vanilla Motif and Motif XP. For Microsoft Windows code generation there are two extra filename filters:

```
visu.visualC++Filter:*.cpp
visu.visualC++StubsFilter:*.cpp
```

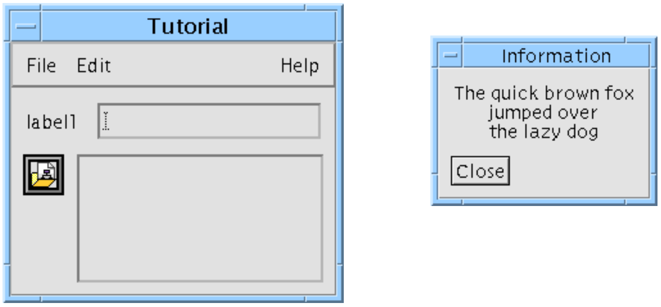
If you wish to share code between the two platforms, you might consider changing the filename filters for the two different flavors so that they are the same. See "File names" on page 384 for more details about points to bear in mind when naming files intended for both platforms.

CHAPTER 12

Creating a Microsoft Windows and Motif Application

Introduction

This chapter shows you how to produce the simple application shown in Figure 12-1 on both Motif and Microsoft Windows. In addition, “Single Sourcing” on page 413 at the end of this chapter looks at the how you might share callback files between UNIX and Microsoft Windows.



The quick brown fox
jumped over
the lazy dog

FIGURE 12-1 Final Application

Starting Your Design

This tutorial shows you how to put together a main application window and a sub-dialog. As well as building the user interface, it uses links, callbacks and sets resources, including pixmaps. It even shows you how to pop up a menu in a drawing area. All of this will be ported to Microsoft Windows.

It is assumed that you are familiar with the basic use of Sun WorkShop Visual. For this reason, a design file containing the completed hierarchies for this tutorial is included with Sun WorkShop Visual. You can either open this file, as explained in Step 2, or follow the instructions to build the hierarchy yourself.

1. **Start Sun WorkShop Visual in Microsoft Windows mode.**

   See “Starting in Microsoft Windows Mode” on page 360 if you are not sure how.

2. **If you wish to save time by not building the hierarchies yourself, open the following file and skip ahead to Step 18 on page 396:**

   ```
   $VISUROOT/src/examples/tutorial/windows.xd
   ```

   where VISUROOT is the root directory of your installed Sun WorkShop Visual.

Note – You only need to do Step 3 through to Step 17 if you wish to build the hierarchy yourself.

---

3. **Start with an application shell. Add a form to it and put a menubar, a button, a scrolled window, a label and a text field inside the form.**

   This hierarchy is shown in Figure 12-2.

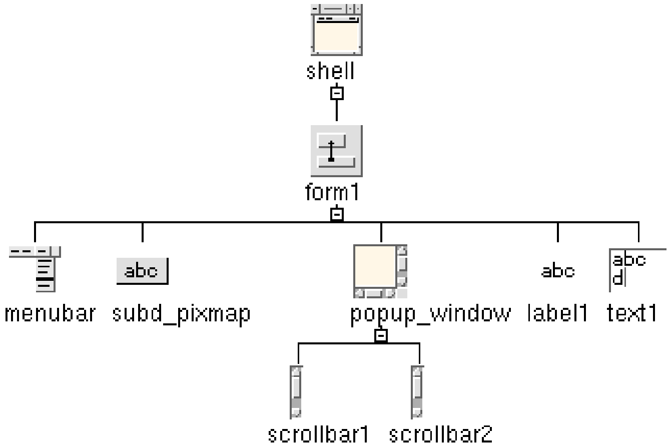


FIGURE 12-2 The Initial Hierarchy

4. **Name the shell "shell", the menubar "menubar", the scrolledwindow "popup_window" and the button "subd_pixmap".**

   You do not need to name any other widgets - names of the type [widget-class][number] (e.g. "button1") are the default names assigned by Sun WorkShop Visual.

5. **To the menubar, add three cascade buttons. Name them "file_menu, "edit_menu" and "help_menu" respectively.**

6. **To the "file_menu" cascade button, add a menu. Add two buttons, a separator and another button to this menu.**

7. **Name the last button added to the menu "exit_b".**

8. **To the "edit_menu" cascade button, add a menu. Add three buttons to this menu.**

9. **To the "help_menu" cascade button, add a menu. Add two buttons to this menu, naming the first one "subd_b".**

   The completed hierarchy for the menubar is shown in Figure 12-3.

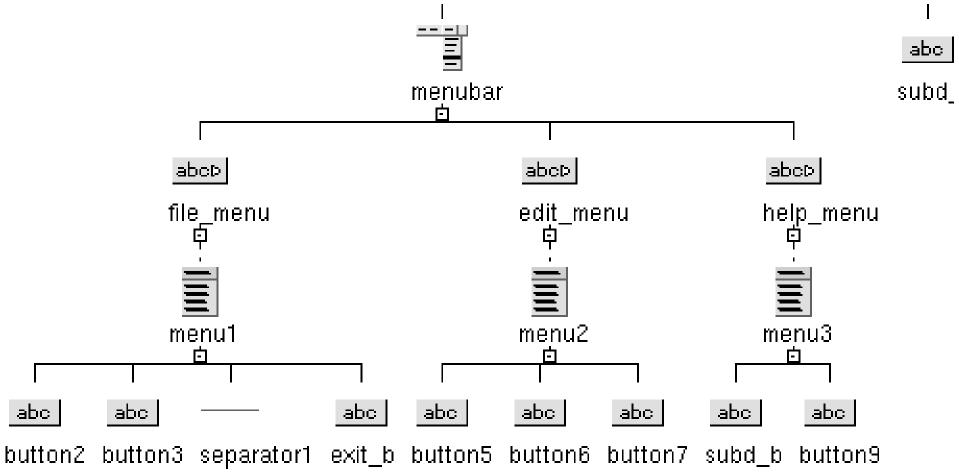


FIGURE 12-3 The Hierarchy Under the MenuBar

10. **To the "popup_window" scrolled window, add a drawing area. Name it "draw_area".**

11. **To the drawing area, add a popup menu naming it "popup". Add three buttons to the popup menu.**

    The completed scrolled window hierarchy is shown in Figure 12-4.

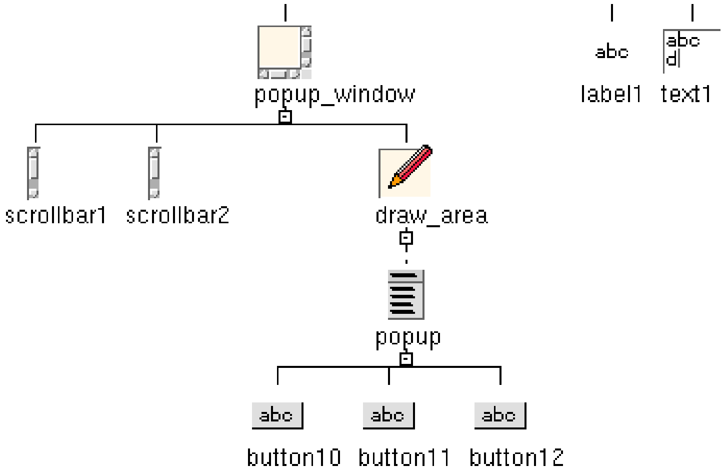


FIGURE 12-4 The Hierarchy Under the ScrolledWindow

12. **Use the Form layout editor to adjust the layout of the widgets inside the form so that the dynamic display looks as shown in Figure 12-5.**

    You will need to move the widgets to the right place and specify attachments. If you are not sure how to do this, read Chapter 4, “The Layout Editor”, starting on page 97, first.

    Although the Form widget will not be generated into the Microsoft Windows code (we have not made it a C++ class), attachments and positions are calculated and handled by Sun WorkShop Visual in the generated code. In this way the resize behavior will be preserved. See “Manager Widgets and Layout” on page 373 for more details.

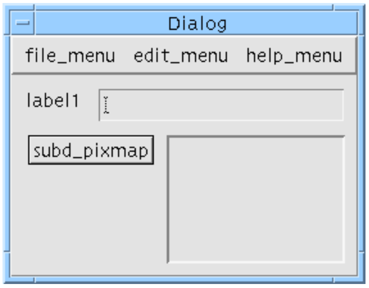


FIGURE 12-5 The Main Window Dynamic Display

13. **Save your design into a file named** `MyWinApp.xd`**.**
14. **Add another shell to your design - this time a dialog shell. Name it “sub_shell”.**
15. **Add a form to the shell and name it “sub_form”.**
16. **Add a label and a button to the form. Name the button “close_b”.**

    The completed hierarchy for this sub-dialog is shown in Figure 12-6.

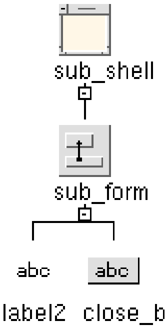


FIGURE 12-6 The Sub-Dialog Hierarchy

17. **Use the Form Layout Editor to layout the widgets so that the button is at the bottom of the dialog and the label grows when the dialog is resized, as shown in Figure 12-7.**

    You will need to move the widgets to the right place and specify attachments. To make the label grow, for example, attach the bottom of the label to the top of the button and attach all other edges to the sides of the form. If you are not sure how to do this, read Chapter 4, “The Layout Editor”, starting on page 97, first.

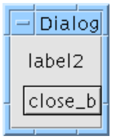


FIGURE 12-7 The Sub-Dialog

18. **Save your design.**

    Do this even if you have opened the supplied design file in Step 2 on page 392. Then you will have your own local copy. Name it `MyWinApp.xd`.

Adding Callbacks

Here we shall add a callback method for the close button which will close the sub-dialog, and a callback method on exit_b in the file menu of the main application shell.

1. **Select the close button, close_b.**
2. **Display the Callbacks dialog and select the Activate callback from the list of callback types.**
3. **Check that the option menu underneath the list of defined callbacks says “Method name”.**

    You will be generating C++ so you need to check that you are adding methods.

4. **Type `DoClose` into the text box and press the “Add” button.**
5. **Close the Callbacks dialog.**
6. **Select the shell named “shell” in the window holding area.**
7. **Select the button named “exit_b” underneath the file menu.**
8. **Display the Callbacks dialog and select the Activate callback from the list of callback types.**

9. **Check that the option menu underneath the list of defined callbacks says "Method name".**
10. **Type DoExit into the text box and press the "Add" button.**
11. **Close the Callbacks dialog.**
12. **Save your design.**

Note – For a Motif-only application you would not need to add a callback to close the application from a button - instead you could set the AutoUnmanage resource in the about_form resources to "Yes". For Microsoft Windows, however, you need to add a callback.

Adding a Link

You are going to add a link to display the sub-dialog from the main window. Sun WorkShop Visual is able to generate code to implement links on both Motif and Microsoft Windows. See "Using Links" on page 372 for restrictions on Microsoft Windows which should be taken into account when creating links. Read "Links" on page 183 if you are not sure how to set up links in a dialog.

1. **Select "subd_b" in the menu called "help_menu" in the main application shell "shell" and display the Edit Links dialog.**

   Do this either by pressing the button on the toolbar or by selecting "Edit Links" from the Widget menu. Figure 2-2 on page 17 shows you where to find the Edit Links button on the toolbar.

   Note that the Add button is pink when a menu button is selected as the destination widget. Show, Hide, Manage and Unmanage links are only effective for menu buttons on Motif; Enable and Disable links are effective on menu buttons in both Motif and Microsoft Windows. However, the menu button currently selected is not the destination widget for this exercise.

2. **Select "sub_shell" in the window holding area.**

   The text box labelled "Widget" in the Links dialog now contains the name of the about dialog shell, "sub_shell".

3. **Select the "Show" link in the Links dialog and press the "Add" button to add the Show link to "sub_shell".**
4. **Close the Edit Links dialog.**

   You are now going to add the same link from "subd_pixmap" in the main application shell.

5. **Go back to the main application shell, “shell”.**

   Select the shell from the window holding area.

6. **Select “subd_pixmap” and display the Edit Links dialog again.**

   You need to redisplay the dialog so that the selected widget is the “source” widget. Otherwise, it is a “destination” widget.

7. **Select “sub_shell” in the window holding area again.**

8. **Select the “Show” link in the Links dialog and press the “Add” button to add the Show link to “sub_shell”.**

9. **Close the Links dialog and save the design.**

Popup Menu

Adding a popup menu to a DrawingArea widget is useful whichever language you intend to generate. The next stage of the tutorial shows you how to do this. You will have to write a small amount of extra code as Sun WorkShop Visual does not automatically generate code to popup menus. Motif and Microsoft Windows have different ways of popping up a menu so your code will have to be different for each platform. The code is added much later in the tutorial, first you need to set up the callback.

1. **Select “draw_area” from inside the ScrolledWindow, “popup_window”, in the main application shell.**

2. **Display the Callbacks dialog.**

3. **Select “Input” from the list of callback types and add a callback method called `DoInput`.**

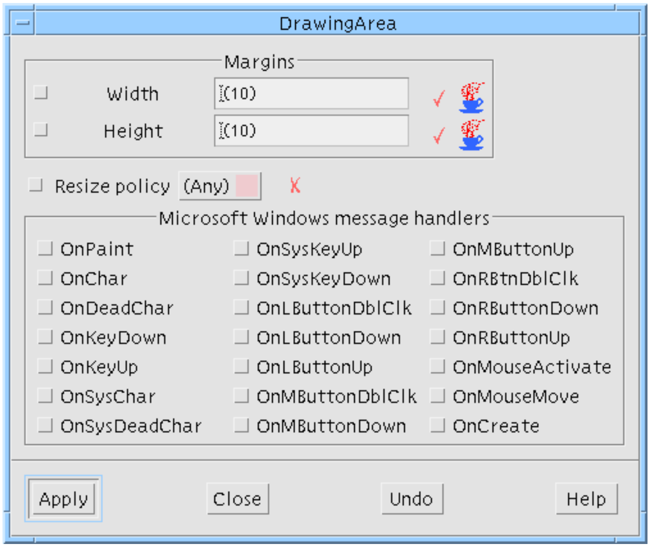
   The Input callback type is marked with an asterisk (*), indicating that this callback will have no effect on Microsoft Windows.

4. **Close the Callbacks dialog.**

5. **Display the DrawingArea resource panel, shown in Figure 12-8.**

   The quickest way to do this is to double-click over the drawing area.

   The Motif XP library does not attempt to emulate the Microsoft Windows drawing or input model, hence the list of Microsoft Windows target message handlers.



DrawingArea

Margins

□ Width (10)

□ Height (10)

□ Resize policy (Any)

FIGURE 12-8 DrawingArea Resource Panel

6. **Set the OnRButtonDown toggle, press "Apply" and close the resource panel.**

   This will generate an appropriate Microsoft Windows message handler, which will be called in response to the corresponding Microsoft Windows message, and a matching callback stub.

   "Filling in the Stubs" on page 406 of this chapter explains how to fill in the stubs to popup the menu.

7. **Save the design.**

Setting Resources

Sun WorkShop Visual generates resource files for Motif and Microsoft Windows. Motif, however, allows a far greater range of control over its widgets. Although there are resources in Microsoft Windows, they are limited in comparison with Motif/X resources. Microsoft Windows resources are compiled into the executable file so, unlike Motif, changing a resource on Microsoft Windows means re-compiling the application.

Setting Label Resources

In this example application, we have string, pixmap and keyboard resources. First of all, we shall set the strings of the labels, buttons and shells.

1. **Select the form which is the child of the main application shell.**
2. **Display the form's resource panel.**
3. **Set the dialog title to "Tutorial" by typing into the text field labelled "Title".**
4. **Press "Apply" and then close the resource panel.**
5. **Display the resource panel of the "file_menu" cascade button and give it the label string "File". Press "Apply".**
6. **Do the same for the other two cascade buttons so that "edit_menu" displays the label "Edit" and "help_menu" displays the label "Help".**

   This is shown in Figure 12-9.

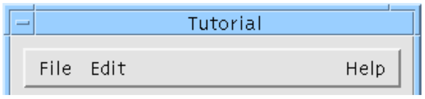


FIGURE 12-9 MenuBar Labels and Shell Title

7. **Set the labels of the buttons in "file_menu" to "New", "Open" and "Exit".**
8. **Set the labels of the buttons in "edit_menu" to "Cut", "Copy" and "Paste".**
9. **Set the labels of the buttons in "help_menu" to "About..." and "Help".**

   The three menus are shown in Figure 12-10.

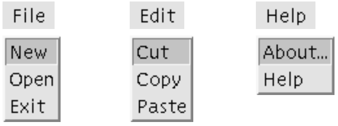


FIGURE 12-10 Menu Item Labels

10. **Select the first button in the popup menu child of the drawing area and display its resource panel.**
11. **Set the button's label to "Cut".**
12. **Select each of the other two buttons in the popup menu and set their labels to "Copy" and "Paste" respectively.**

13. **Select the dialog shell “sub_shell” in the window holding area.**

    The design area now shows the hierarchy for this sub-dialog.

14. **Select the label and display its resource panel.**

15. **Set the string of the label widget to display a few lines such as:**

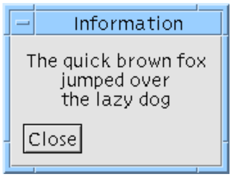
    ```
    The quick brown fox
    jumped over
    the lazy dog
    ```

16. **Center the string in the label by going to the “Settings” page of the resource panel and changing the “Alignment” option menu to “Center”. Press “Apply”.**

17. **Set the label of the button to “Close”.**

18. **Set the title of the sub-dialog to “Information”**

    Do this by typing into the text box labelled “Title” in the resource panel of the Form which is the child of the shell. The completed dialog is shown in Figure 12-11.



The quick brown fox
jumped over
the lazy dog

FIGURE 12-11 Completed Sub-Dialog

19. **Save your design.**

Generating String Resources

All of these labels are string resources. When you generate the code for the Motif version of your design you can decide whether the resources should go into the code or into the resource file. For the Microsoft Windows version all string resources are automatically generated into the source code.

Using Pixmaps

If you are using pixmaps in your design, Sun WorkShop Visual automatically converts them to Windows bitmaps when you generate code. To illustrate this, you are going to give one of the buttons a pixmap label.

1. **Select the button named “subd_pixmap” in the main application shell and display its resource panel.**
2. **Set the Type Resource of the buttons to Pixmap.**

   You can find this on the “Settings” page of the button resource panel.
3. **Go to the “Display” page of the button resource panel and press the button labelled “Pixmap”.**

   This displays the Pixmap Selector dialog.
4. **In the Pixmap Selector dialog, either select an existing pixmap or press “Edit” to display the Pixmap Editor.**
5. **If you are creating a new pixmap in the Pixmap Editor, make sure that you type a name for it into the text field at the top and press “Bind”. Close the Pixmap Editor.**
6. **In the Pixmap Selector dialog, press “Apply” so that the object name appears in the text field next to the button labelled “Pixmap” in the resource panel.**

   If you are not sure about creating and using pixmap objects, see “Selecting a Pixmap” on page 148.
7. **When you press “Apply” in the resource panel, the pixmap you have specified is shown on the button.**
8. **Save your design.**

Naming the Pixmap

Remember that filenames on MS-DOS (and Microsoft Windows) must be no longer than eight characters before the extension. Sun WorkShop Visual uses the name to which you bind the pixmap in the basename of the file in Microsoft Windows mode, so make sure that you have not specified a name longer than eight characters.

You have now finished the design of your application. The rest of this tutorial shows you how to generate code for the UNIX and for Microsoft Windows and how to build the application on each platform.

Building the Application

Now that you have designed the application user interface in Sun WorkShop Visual, you can generate the Motif and Microsoft Windows code. Having generated the code, you are then ready to build the two applications. Full instructions are provided in this section, including how to fill in the callback stubs.

Controlling the Sources

For this tutorial, you need to generate two separate applications - one for Motif and one for Microsoft Windows. You should generate them into two separate directories to avoid such problems as overwriting and including the wrong header file. This tutorial does not exploit the possibilities of single sourcing, which are discussed in "Single Sourcing" on page 413.

Code Generation for Motif

1. **Display the Generate dialog.**

   The filename fields are primed with names based on your Sun WorkShop Visual save file name.

2. **Check that "C++ Motif XP" is the selected item from the "Language" option menu.**

   At the top of the dialog there is a text area where you can type the name of a base directory. All filenames are then relative to this directory. You can use the "Browse" button to find a directory.

3. **Set the base directory to the area where you wish to generate your files.**

   Remember that you should generate the Motif XP and the Microsoft Windows MFC code into separate directories to avoid confusion.

4. **Check that the "Generate" toggle is set for the "Code" file.**

5. **Open the Code Options dialog.**

   Press the button labelled "Options" next to the "Code" text box.

6. **Set the "Include Header File" toggle**

7. **Make sure that the "Include Motif Header Files" toggle is set.**

8. **Press "Ok".**

   This saves your changes and closes the Code Options dialog.

9. **Check that the "Generate" toggle next to "Externs" is set.**

10. **Set the "Generate" toggle next to the "Stubs" file.**

11. **Make sure that the "Generate" toggle next to the "Main Program" text box is set.**

12. **Display the Generate Options dialog.**

    Do this by pressing the button labelled "Options" at the bottom of the Generate dialog.

13. **Check that the "Links" option menu is set to "Generated to Code".**

14. **Change the "Strings" option menu to "Code" and check that all other resource types are set to "Code".**

    For the sake of simplicity, all the resource settings are going to be "hard-coded" for this tutorial. For a real application, you would normally generate them into a separate resource file.

15. **Press "Ok".**

    This saves your changes and closes the Generate Options dialog.

16. **Set the "Generate" toggle next to "Makefile".**

17. **Press "Generate" in the Generate dialog.**

    Your files are generated into the chosen directory.

18. **Save the design.**

    This saves any settings in the Generate dialog.

The Makefile

You may have to edit the Makefile in order to access the Motif XP code, depending on how Sun WorkShop Visual has been installed and configured. Check that the `XPCLASSLIBS` and `CCFLAGS` definitions access the Motif XP library and include files. VISUROOT is the path to the root of the Sun WorkShop Visual installation directory:

```
XPCLASS = $(VISUROOT)/src/motifxp

XPCLASSLIBS = $(XPCLASS)/lib${ABIDIR}/libmotifxp.a

GEN_CFLAGS=-I. ${XINCLUDES} -I${XPMDIR} ${GROUP_COMPILEFLAGS}

CCFLAGS=${CFLAGS} ${ABICCFLAGS} -I${XPCLASS}/h ${GROUP_COMPILEFLAGS}
```

Code Generation for Microsoft Windows

The steps to generate the code for Microsoft Windows are almost the same as for Motif. Sun WorkShop Visual remembers a different set of files for each flavor so you can use the toolbar flavor menus and code generation buttons once you have specified the name of the files for the different flavors.

Remember that, on the PC, a filename must be eight characters or less before the extension. Some IDEs, including Visual C++, complain if they encounter a source file containing C++ code which has been given only a '`.c`' extension. You can specify "`.cpp`" or "`.cxx`".

1. **Display the Generate dialog if you do not still have it open on the screen.**

2. **Check that "C++ Microsoft Windows MFC" is the selected item from the "Language" option menu.**

3. **Set the base directory at the top of the Generate dialog to a directory where you wish to generate your files.**

   Remember that you should generate the Motif XP and the Microsoft Windows MFC code into separate directories to avoid confusion.

4. **Check that the "Generate" toggle next to the "Code" file is set.**

5. **Open the Code Options dialog.**

   Press the button labelled "Options" next to the "Code" text box.

6. **Set the "Include Header File" toggle.**

7. **Press "Ok".**

   This saves your changes and closes the Code Options dialog.

8. **Check that the "Generate" toggle is set for the "Externs" and "Stubs" files.**

9. **Make sure that the "Generate" toggle next to the "Main Program" text box is set.**

10. **Set the "Generate" toggle next to the "Microsoft Windows resources" file.**

    This file will contain the dialog templates. When this is set, Sun WorkShop Visual converts pixmaps into Windows bitmaps - one per file, as described in "Pixmaps, Bitmaps, and Icons" on page 376.

11. **Display the Generate Options dialog.**

    Press the button labelled "Options" at the bottom of the Generate dialog.

12. **In the Generate Options dialog, check that the "Generate as Resources" toggle is set.**

    This makes Sun WorkShop Visual generate dialog templates which are Microsoft Windows resource files, as described in "Generating Dialog Templates" on page 386.

13. **If you are using MFC 4 or 5, set the "MFC 4 Enhancements" toggle.**

    This gives you the 3D look and feel.

14. **Check that the option menus for "Links" and all resources are set to "Generated to Code".**

15. **Press "Ok".**

    This saves your changes and closes the Generate Options dialog.

Note – You do not need to generate a makefile for the Microsoft Windows code because the files will be built in a different way. This is described in "Compiling the Microsoft Windows Version" on page 409.

16. **In the Generate dialog, press "Generate".**

    A message is displayed informing you that a bitmap file for your pixmap has been generated. The basename of the file is the name of the pixmap object.

The Generated Files

When you generate MFC, you will find that extra files in addition to the code, stubs and header files are generated. These are:

1. Bitmap files. If you have created any pixmap objects, they are converted to Windows bitmap files, a file per pixmap. These files have a ".ico" suffix.
2. Project files. These are Visual C++ version 5 (and above) files which, similar to a Makefile, contain all the information needed to load the application into Visual C++ where it can then be built. These files have a ".dsw" and ".dsp". More about these files is given in "Project Files" on page 387.

Japanese Text

If the generated code contains Japanese text, you need to post-process the code with a filter utility, xdtosj, which is provided as part of the Sun WorkShop Visual release, before transferring it to a PC. See "Use of Japanese Font" on page 388 for more information.

Filling in the Stubs

You have two stubs files - one for MFC and one for Motif. Because you generated Motif XP code for your Motif application you could share some of the code. This is because Sun WorkShop Visual's XP library emulates some of the MFC for Motif. For simplicity and speed, however, this tutorial keeps the two separate. For more information on sharing the sources, see "Single Sourcing" on page 413.

The MFC Stubs

There are two callbacks - one to exit the application and one to popup the menu from the drawing area. You should fill in the stubs as shown below:

```
void
shell_c::DoExit ( )
{
```

Add this line

```
    exit(0);
}
```

```
void
sub_shell_c::DoClose ( )
{
```

Add this line

```
    this->ShowWindow(SW_HIDE);
}
```

```
afx_msg void popup_window_c::OnRButtonDown( UINT nFlags, CPoint point
)
{
```

Add these lines

```
    ClientToScreen(&point);
    popup->TrackPopupMenu( TPM_LEFTALIGN|TPM_RIGHTBUTTON,
                                point.x, point.y, this, NULL);
}
```

The Motif Stubs

There are two callbacks in the Motif stubs file to match the two in the MFC stubs file. Although the signature of the exit callback is the same for both platforms, the contents are different.

```
void
shell_c::DoExit ( )
{
```

Add this line

```
    exit (0);
}
```

```
                void
                sub_shell_c::DoClose ( )
                {
Add this line  ──▶   this->ShowWindow(SW_HIDE);
                }

                void
                popup_window_c::DoInput ( )
                {
Add this line  ──▶   popup->TrackPopupMenu(0, 0, 0, this, NULL);
                }
```

Compiling the Application

Having created the design, generated code and filled in the callback stubs, you are now ready to compile the application on both Motif and Microsoft Windows. After the brief section on building your application on Motif, there is a description of the steps you will need to take to build your application on Microsoft Windows.

Step-by-step instructions for building the application using Visual C++ versions 4.0 and 5 on Microsoft Windows are given in “Compiling the Microsoft Windows Version” on page 409. However, you can build the generated code using any C++ development environment capable of integrating the Microsoft Foundation Classes, for example Symantec C++. Refer to “Using Other Applications” on page 412 for general information about configuring Microsoft Windows-based C++ compilers to build the application.

Compiling the Motif Version

All you need to do now is make the application by typing

```
make
```

at the command prompt in the directory where you generated the Motif code. Since you generated a Makefile, all the files in your application are automatically built.

Before running the application, be sure to arrange for the resource file to be read, as explained in “Setting up the X Resource File” on page 215.

Compiling the Microsoft Windows Version

If you are using Visual C++ Version 5, compiling your application on your PC is very simple because Sun WorkShop Visual generates full Visual C++ project files for you. If you are using earlier versions of Visual C+ you will need to create the project. Although this is not so convenient, it does not take long and only needs to be done once. Full instructions are given below for using Visual C++ Version 5 and Visual C++ Version 4.0. If you are using a version of Visual C++ earlier than 4.0 and you need help on creating a project, contact Sun WorkShop Visual support. "Using Other Applications" on page 412 gives you some tips on compiling your application if you are not using Visual C++.

Brief Notes on Visual C++

Visual C++ is an IDE (Integrated Development Environment) to help you develop applications to run on Microsoft Windows. It comprises a number of tools, including a compiler, debugger and various editors. It is a useful tool for building, debugging and controlling the sources of an application. Note, however, that any changes you make to your code in Visual C++ cannot be taken back into Sun WorkShop Visual. Visual C++ uses the concept of a *project* to keep track of the files in your application. You always need a project when you use Visual C++. For Version 5, Sun WorkShop Visual generates the project files for you.

Note – PC-NFS, available from SunSoft, is a tool designed to make sharing files between your PC and your Solaris system easy.

Visual C++ Version 5

Sun WorkShop Visual generates fully configured project files for Visual C++ version 5 and above. These are the files ending with a suffix of ".dsw" and ".dsp". Take both of these files to your PC along with **all** the MFC source files (this includes the code files, resource file, any bitmap files and header files).

1. **On your PC, under Windows, double-click over the ".dsw" file generated by Sun WorkShop Visual**.

   Doing this instructs Sun WorkShop Visual to open Visual C++, with the project ready loaded.

2. **Select "Build <projectname>.exe" from the Build menu.**

   If Visual C++ encounters any errors, these are displayed and the compiler stops compiling. You may now press F4 to ask Visual C++ to open the file containing the error and locate you at the relevant point in the file. Subsequent presses of F4 allow you to move through the list of errors, opening files as necessary. Double clicking on an error will also open the relevant file and move to the part of the file where the error was detected. The windows opened onto these files are full text editing windows. You can also view and edit a file in this way by selecting Open... from the File menu.

3. **Once the application has built successfully, you can try it out. Select "Execute <projectname>.exe" from the Build menu.**

Visual C++ Version 4.0

If you are using Visual C++ Version 4, you cannot use the project files generated by Sun WorkShop Visual. The following takes you through the steps required to create the project.

1. **Take all the MFC source files to your PC.**

   You do not need to take the ".dsp" and ".dsw" files as these are only relevant to Visual C++ Version 5 and above.

2. **Start Visual C++.**

3. **Select "New" from the File menu.**

   A list of new items which can be created is then displayed as shown below:

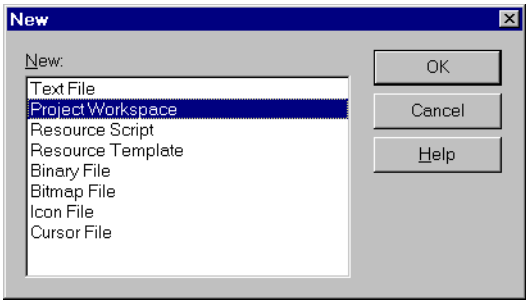
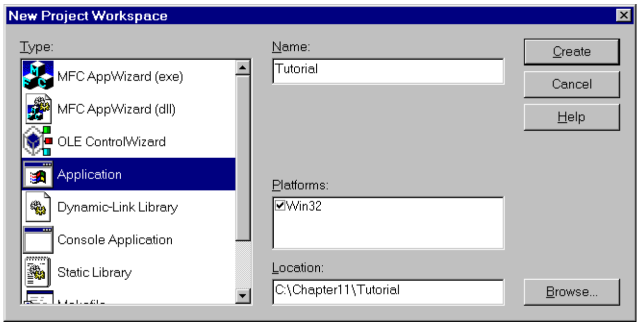


FIGURE 12-12 New Items List

4. **Select "Project Workspace" from the items list and Ok the dialog.**

   The New Project Workspace dialog is then displayed.



Name: Tutorial

Platforms: ☑Win32

Location: C:\Chapter11\Tutorial

FIGURE 12-13 New Project Workspace Dialog

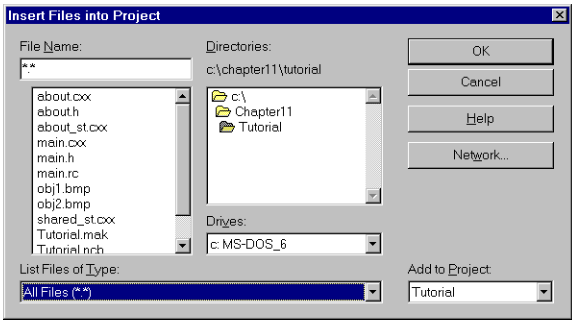
5. **Select “Application” from the Type list and enter a name (such as “Tutorial”) in the “Name” field.**

6. **Press the “Create” button to complete your actions.**

   You are then presented with the main Visual C++ dialog. Next, you must now populate your project with the files you wish to build.

7. **Select “Files into project” from the Insert menu.**

   The following dialog is displayed:



File Name: *.*

Directories: c:\chapter11\tutorial

Drives: c: MS-DOS_6

List Files of Type: All Files (*.*)

Add to Project: Tutorial

FIGURE 12-14 Insert Files Dialog

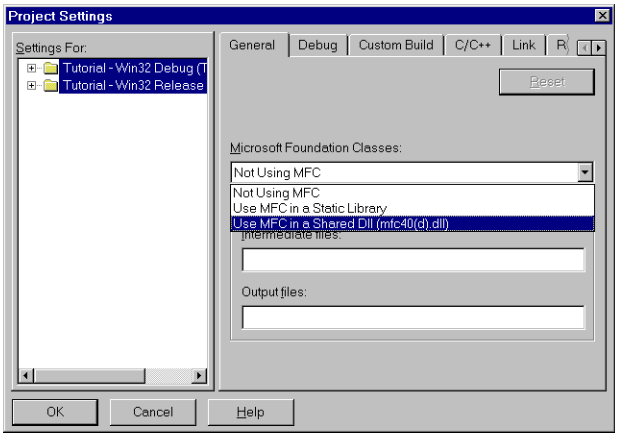
8. **Locate the directory where you have placed the files you copied to the PC.**

9. **Highlight the files required by clicking on the first in the list then press shift and then click on the last of the files in the list.**

10. **Press the Ok button to add the selected files to the project.**

11. **Select the “Settings” option from the Build menu.**

    The Project Settings dialog is then displayed, as shown below:



Microsoft Foundation Classes: Not Using MFC

- Not Using MFC
- Use MFC in a Static Library
- Use MFC in a Shared Dll (mfc40(d).dll)

FIGURE 12-15 Project Settings Dialog

12. **Select the "Use MFC in a Shared Dll..." option from the Microsoft Foundation Classes option menu as shown above and then Ok the dialog.**

    Finally, to build your application:

13. **Select "Build Tutorial.exe" from the Build menu.**

    If Visual C++ encounters any errors, these are displayed and the compiler stops compiling. You may now press F4 to ask Visual C++ to open the file containing the error and locate you at the relevant point in the file. Subsequent presses of F4 allow you to move through the list of errors, opening files as necessary. Double clicking on an error will also open the relevant file and move to the part of the file where the error was detected. The windows opened onto these files are full text editing windows. You can also view and edit a file in this way by selecting Open... from the File menu.

14. **Once the application has built successfully, you can try it out. Select "Execute <projectname>.exe" from the Build menu.**

Using Other Applications

The code generated by Sun WorkShop Visual is MFC code. It has not been generated for any specific Microsoft Windows application. The following lists the important points to bear in mind when building your user interface on Microsoft Windows *if you are not using Visual C++*. This section applies to all compilers.

1. **You will need a C++ compiler and the Microsoft Foundation Class (MFC) include files and libraries installed on your system.**

2. **Configure the build tool to build a Microsoft Windows .EXE file.**

3. **Make sure the compiler has a valid include path to the MFC header files.**
4. **Make sure the compiler has a valid include path to the subdirectory containing the Sun WorkShop Visual-generated source files.**
5. **Compile the code using large memory model settings.**

   Make sure the linker links in the MFC libraries or DLLs.

Single Sourcing

Using Sun WorkShop Visual's Motif XP library gives you the potential to share some of your callback code between Motif and MFC. Appendix B, "Motif XP Reference", starting on page 835 lists the classes which have been emulated for Motif. Use this appendix to find out which calls to the MFC toolkit you can use for both platforms.

To use the XP library, you should choose "Motif XP" as the language to generate for your Motif application in the Generate dialog. When generating code, you should remember that only a suffix of ".cxx" or ".cpp" is acceptable to both UNIX and Windows platforms. The following list of steps uses pre-built designs and a stubs file which has already been filled in to illustrate how you can use the Motif XP to write a callbacks file which can be used on both Motif and Windows. The files required for this are found in $VISUROOT/src/examples/tutorial where VISUROOT is the install directory of your Sun WorkShop Visual.

1. **Start Sun WorkShop Visual in Windows mode.**

   See "Starting in Microsoft Windows Mode" on page 360 if you need more information on this.
2. **Choose "Open" from the File menu and read in this file**:

   `$VISUROOT/src/examples/tutorial/singlesource.xd`
3. **Check that you have the design shown in Figure 12-16.**

   It is a simple dialog consisting of two option menus, two toggles and a text field.

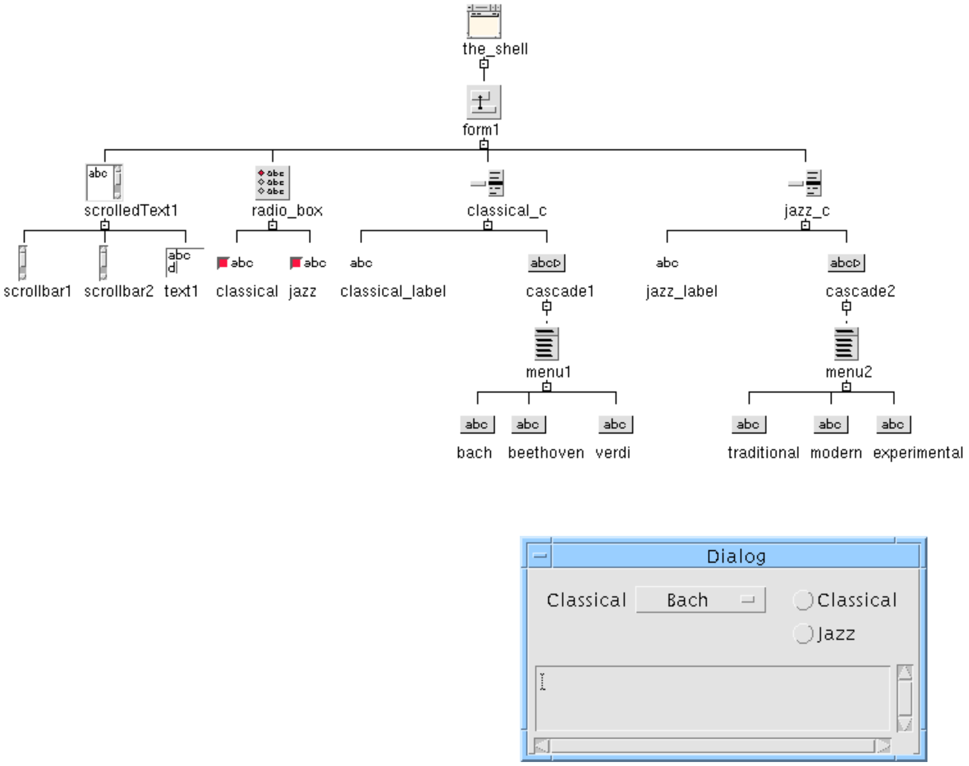


FIGURE 12-16 Design Hierarchy and Dynamic Display

4. **Display the Generate dialog and choose "Motif XP" as the first language to generate.**

   Make sure that you have a target directory for the Motif-only sources.

5. **Generate the Code file, Externs file, Main code file, Makefile and Stubs file. Use the names already shown in the Generate dialog.**

   You are going to overwrite the Stubs file with a pre-configured one, but you need Sun WorkShop Visual to create a dependency for it in the Makefile.

6. **Change the "Language" to "MFC". Remember to change the target directory as well.**

7. **Generate the Code file, Externs file, Main code file and Windows Resource file but not the Stubs file.**

   You are going to use the same pre-configured Stubs file mentioned above, but you need Sun WorkShop Visual to create a dependency for it in the Visual C++ project files.

8. **Copy over the following stubs file into your Motif XP directory:**

   ```
   $VISUROOT/src/examples/mfc/singlesource_stubs.cpp
   ```

   You will just need to remember to copy it over to your PC along with the MFC source you just generated.

   You can look at the Stubs file in a text editor. There are three callbacks:

   1. **JazzChanged**. This one checks the state of the “Jazz” toggle and displays the “Jazz” option menu if it is set, hiding the “Classical” option menu.
   2. **ClassicalChanged**. This one checks the state of the “Classical” toggle and displays the “Classical” option menu if it is set, hiding the “Jazz” option menu.
   3. **DoSetText**. This fetches the selected item from the visible option menu and puts the text into the text field.

   All of this code can be shared by the Motif XP application and the MFC application.

9. **To compile the Motif application, simply make sure you are in your motif directory and type:**

   ```
   make
   ```

   at the command prompt. See Chapter 7, “Generating Code”, starting on page 207 for more information on generating, compiling and running code.

10. **Once it has built, run the application on Motif checking that your callbacks work. Try setting the two toggles and making selections from the option menus.**

11. **To build the MFC application:**

    a. **Take all of the files generated into your MFC directory to your PC.**

       If you are using Visual C++ version 5, remember to take the “.dsw” and the “.dsp” files. These are Visual C++ project files.

    b. **Take the pre-configured Stubs file from the Motif XP directory to your PC, overwriting the *first* one, which was only generated to force the correct dependencies.**

    c. **If you are not using Visual C++, see “Using Other Applications” on page 412 for points to remember when compiling on your PC.**

    d. **On your PC, open the “.dsw” file in Visual C++ as the project workspace.**

       This assumes you are using Visual C++ Version 5. If you are using an earlier version or a different application, refer to “Compiling the Microsoft Windows Version” on page 409.

    e. **Build the project.**

    f. **Run the project.**

g. **Check that your callbacks work - try setting the toggles and making selections from the option menus.**

Other Ways to Share Source Between Platforms

There are two other options you could consider for single sourcing of callback files:

1. Using Java

2. Using Get/Set Smart Code

The Java programming language gives you platform independence. Sun WorkShop Visual can generate your design in Java code. In addition, you can save your design in a format ready for importing into Visaj, the Java application builder. Chapter 10, “Designing for Java”, starting on page 313 explains all about creating Java applications from your design.

Get/Set Smart Code gives you a toolkit-independent “wrapper” around widgets that you are interested in. Full details, including a short tutorial, are given in Chapter 16, “Get/Set Smart Code”, starting on page 485.

CHAPTER 13

Design Tools

Introduction

Sun WorkShop Visual includes a number of tools which assist you in your task of building a user interface and linking it to your application. This chapter describes AppGuru and Sun WorkShop Visual Capture. These tools are powerful stand-alone utilities which work in conjunction with Sun WorkShop Visual. They provide a way of starting a design - AppGuru gives you the basis of a new design while Sun WorkShop Visual Capture gives you the design of an existing application.

If you wish to find out more information on other tools in Sun WorkShop Visual, see the following chapters and sections:

- Chapter 14, “Sun WorkShop Visual Replay”, starting on page 433. This chapter provides information on Sun WorkShop Visual Replay which gives you the ability to record and play back the use of an application.
- Chapter 23, “User-Defined Widgets”, starting on page 627. This provides information on the tool visu_config which allows you to use your own widgets in Sun WorkShop Visual.
- “Editing Pixmaps” on page 148 - for more information on creating and editing pixmaps.
- “Setting Fonts” on page 139 - for more information on selecting fonts and font sets.
- “Setting Colors” on page 135 - for more information on choosing colors.

AppGuru

AppGuru provides a fast way to create a standard interface by giving you access to reusable designs, called *templates*. You can create your own templates and make them available to other users. Templates are created and stored in the AppGuru Designer dialog. When a template is selected from this dialog, the design it describes is automatically added to your session of Sun WorkShop Visual.

Display the AppGuru Designer dialog by selecting “AppGuru Designer” from the Tools menu or by pressing the AppGuru button on the toolbar. This button is shown in Figure 13-1.

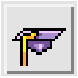


FIGURE 13-1 AppGuru Button on Toolbar

Using AppGuru templates has the following benefits:

1. Styles which must be used within a company or for a particular application are enforced from the beginning.

2. Designs are pre-configured for convenience and to avoid errors.

3. The basic design of dialogs which are very similar can be re-used.

AppGuru Designer

With AppGuru Designer you can:

1. Create new templates “from scratch”.

2. Edit existing templates.

3. Prime a new template from your current design.

4. Select a template to add to your current Sun WorkShop Visual session.

An optional description field in the template definition allows you to provide some information to help you and other users understand what the selected template provides, before adding it to a design.

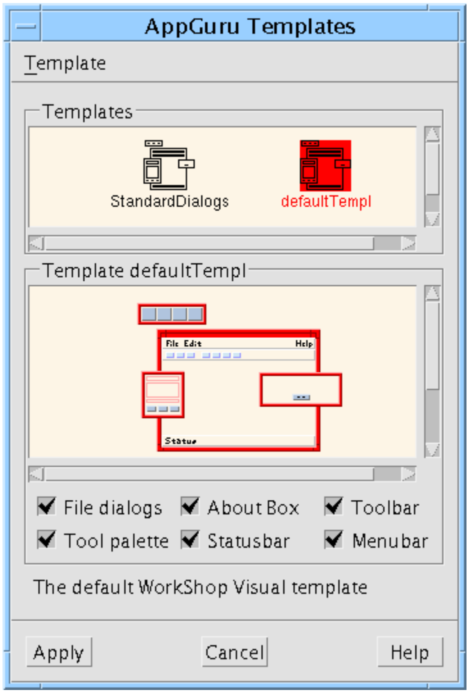
AppGuru Templates

Selecting “AppGuru Designer” from the Tools menu displays the AppGuru Templates dialog, shown in Figure 13-2. This shows all currently available templates. You can load single templates (or all templates in the specified directory) into the AppGuru Templates dialog by selecting the appropriate command from the Template menu. You may also “Unload” a template. This simply removes the template from the AppGuru Templates dialog, any files associated with it remain in place.

Select a template and press “Apply” to add the design described by a template to your existing design.

A template is made up of “components”, each of which may consist of any number of widgets. A component in AppGuru refers to the elements of the template which may be selected and deselected from the AppGuru Template dialog. The widgets of a component define the appearance and behavior of the component.

The templates are shown in the top portion of this dialog as “thumbnail” sketches. As you select each of these, a list of the components defined for that template is shown in the lower portion of the dialog, allowing you to “switch off” any components you do not wish to be added to your design. A single component can represent several widgets in your design.



☑ File dialogs ☑ About Box ☑ Toolbar
☑ Tool palette ☑ Statusbar ☑ Menubar

The default WorkShop Visual template

FIGURE 13-2 AppGuru Templates Dialog

Templates are edited by selecting "Edit Template" and created by selecting "New Template" from the Template menu of this dialog. Selecting either of these menu items displays the AppGuru Edit Template dialog, shown in Figure 13-3.

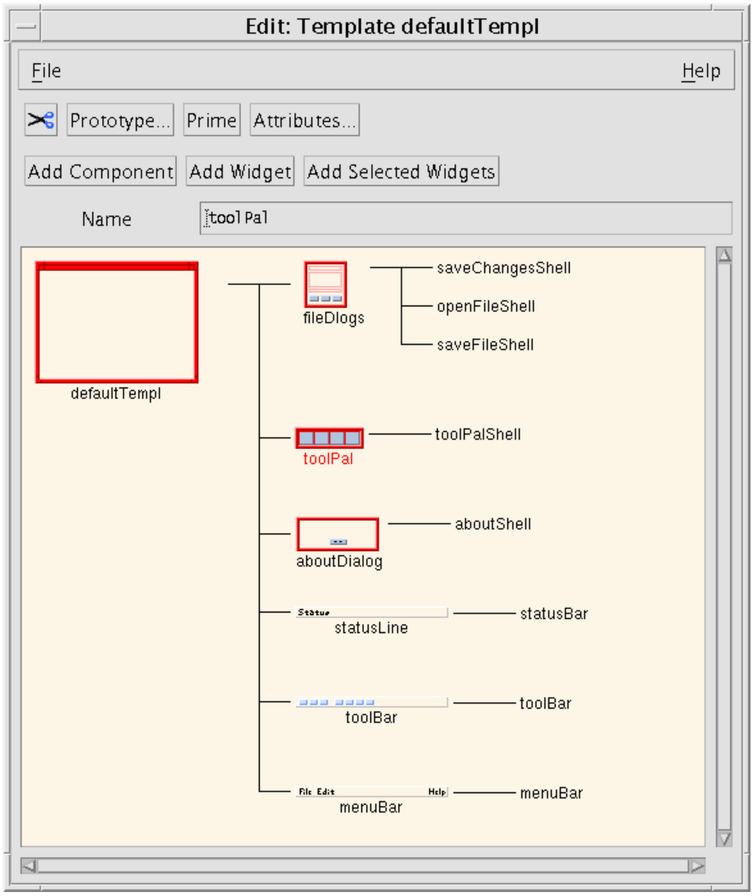


FIGURE 13-3 AppGuru Edit Template Dialog

AppGuru Edit Template Dialog

This dialog allows you to:

1. Edit all the attributes of the template (including all text associated with the template, the location of the design file and the pixmap file).

2. Show how the template under construction will appear in the AppGuru Templates dialog.

3. Add components to the template and remove components from the template.

4. Add widgets to components.

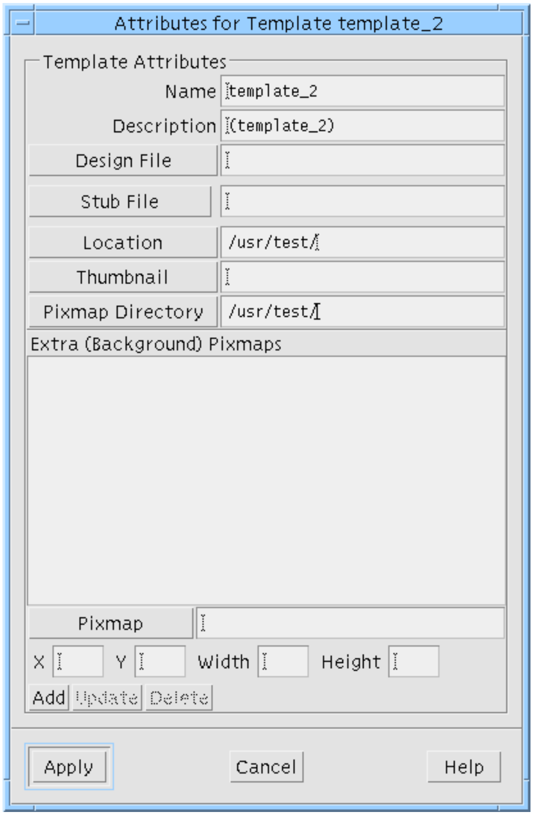
5. Edit the attributes of a component.

6. Save the template.

Each of these is described below.

Template Attributes

Pressing the “Attributes” button when a template is selected, or double-clicking over a template, displays the Template Attributes dialog. This is shown in Figure 13-4.



Name: template_2

Description: (template_2)

Location: /usr/test/

Pixmap Directory: /usr/test/

FIGURE 13-4 Template Attributes Dialog

This dialog allows you to edit the following information:

- The background pixmap. This is the pixmap showing the part of the template which cannot be filtered out; that is, the part left when *all* components are “switched off” in the AppGuru Template dialog.
- The thumbnail pixmap. This is the small sketch displayed at the top of the AppGuru Template dialog.
- The name of the Sun WorkShop Visual save file containing the widgets referred to in this template. Without this information, Sun WorkShop Visual does not have any information on the widgets and therefore cannot display them.
- The location of the template directory. You may wish to keep all your templates in the same directory because Sun WorkShop Visual can load all templates in a directory in one action.
- The directory containing the pixmap files.
- A short description of the template. This is shown in the AppGuru Template dialog when the thumbnail is selected.

Prototype

While you are editing or creating a template, press the “Prototype” button to see how your template will appear when selected in the AppGuru Templates dialog.

Adding and Removing Components

You may add components to a template by:

- Selecting “Prime” from the Template menu.
- Pressing the “Add Component” button.

“Prime” adds all the shells in your current design to the selected template. Each shell is one component of the template. Pressing the “Add Component” button adds one component. To remove components, simply select them and press the Cut button. There is no paste or undo facility.

Adding and Removing a Component’s Widgets

To add a widget to a component:

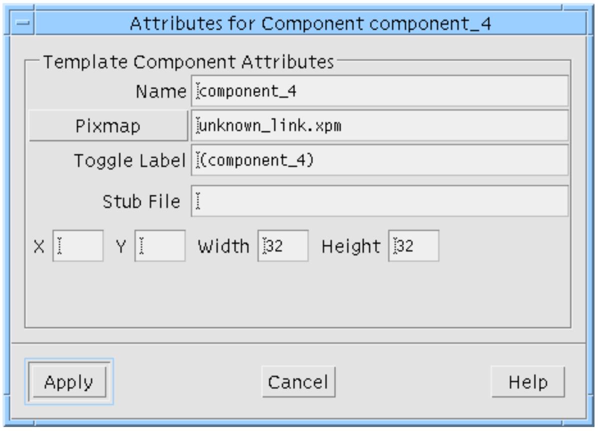
1. **Select the component.**
2. **Select one or more widgets in your Sun WorkShop Visual hierarchy.**
3. **Press the “Add Selected Widgets” button.**

If the widget you wish to add is in another Sun WorkShop Visual save file or it is difficult to select in the current design, press “Add widget”, select the widget and then type the name of the widget in the “Name” text field.

To remove widgets, select them and press the Cut button on the toolbar. Remember that there is no paste or undo facility.

Editing Component Attributes

You can edit various attributes of a component by selecting “Attributes” from the Component menu and entering information into the Component Attributes dialog, which is shown in Figure 13-5.



Attributes for Component component_4

Template Component Attributes

Name: component_4

Pixmap: unknown_link.xpm

Toggle Label: (component_4)

Stub File:

X: Y: Width: 32 Height: 32

FIGURE 13-5 Component Attributes Dialog

In the Component Attributes dialog you can change the name of the component as it is displayed in the AppGuru Editing dialog. The rest of the fields apply to the component as it appears in the AppGuru Templates dialog. The pixmap named in this Attributes dialog is drawn over the template background pixmap at the x and y position given here and with the specified width and height. The toggle label refers to the label of the toggle for “switching off” the component in the AppGuru Templates dialog.

Note – The enhancements to AppGuru are backwards compatible: the former resource-based templates are fully selectable under the new system.

Saving Templates

The File menu in the AppGuru Edit Template dialog contains items to:

1. Save the current template using the current filename, if any.
2. Save the template using a different name.
3. Close the Edit Template dialog.

The saved file is an X resource file. It is this file which is opened by AppGuru. The resource file, in turn, references the Sun WorkShop Visual design file containing the description of the template. Change the name of the design file in the Template Attributes dialog.

Sun WorkShop Visual Capture

Sun WorkShop Visual Capture allows you to *capture* dialogs from a running Motif application and drag them into Sun WorkShop Visual. Whether the application has been “hand-crafted” or designed using a GUI builder, Sun WorkShop Visual Capture can create an “xd” file of the application’s design.

Sun WorkShop Visual Capture is available from the “Tools” menu. You can also use this tool from the command line as described in “Using Sun WorkShop Visual Capture From the Command Line” on page 431.

Before Using Sun WorkShop Visual Capture

For successful operation of Sun WorkShop Visual Capture, the Motif application you wish to record must have been dynamically linked with the Xt library (libXt). On many UNIX implementations, you can find out whether the application has been dynamically or statically linked with libXt by typing:

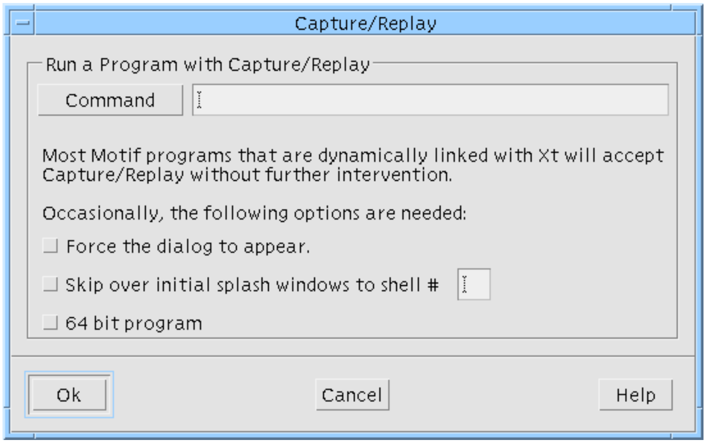
```
ldd AnApplication
```

If the output mentions libXt, the application has been dynamically linked with the Xt library and can be used with Sun WorkShop Visual Capture. If this library is not present, the application has probably been statically linked with the Xt library. You will have to re-link your application with the Xt shared library if you want to use Sun WorkShop Visual Capture.

Sun WorkShop Visual Capture is completely non-intrusive and will not affect either the performance or the behavior of an application.

Running Sun WorkShop Visual Capture

When you select “Sun WorkShop Visual Capture” from the “Tools” menu, a dialog is displayed allowing you to type the name of a Motif application to be captured into the text box next to the button labelled “Executable”. There is also a text box allowing you to type any arguments you wish to be passed to the application. This dialog is shown in Figure 13-6.



Most Motif programs that are dynamically linked with Xt will accept Capture/Replay without further intervention.

Occasionally, the following options are needed:

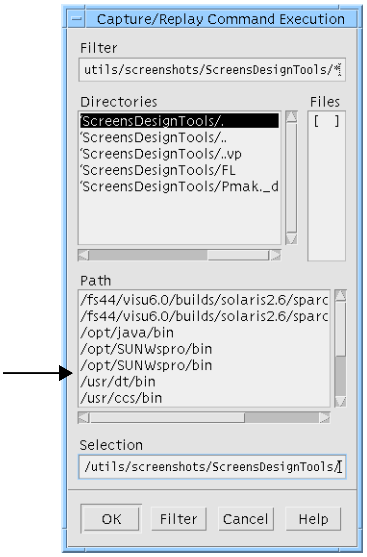
□ Force the dialog to appear.

□ Skip over initial splash windows to shell #

□ 64 bit program

FIGURE 13-6 Capture/Replay Application Prompt

When you press “OK”, your PATH is searched for the named executable. If it is not in your PATH or you are not sure where to find the application, press the button labelled “Executable”. This produces a file selection box containing an extra scrolled list, as shown in Figure 13-7. Each item in this list is a directory from your PATH. This path list, however, is the path set up for you when you run Sun WorkShop Visual and may contain some extra directories required by Sun WorkShop Visual. When you exit Sun WorkShop Visual your PATH is the same as it was before running the application. When you select a directory from this list, any files in the directory are shown in the scrolled list labelled “Files”.



Filter: utils/screenshots/ScreensDesignTools/*

Directories:

'ScreensDesignTools/.
'ScreensDesignTools/..
'ScreensDesignTools/..vp
'ScreensDesignTools/FL
'ScreensDesignTools/Pmak._d

Files: [ ]

Path:

```
/fs44/visu6.0/builds/solaris2.6/sparc
/fs44/visu6.0/builds/solaris2.6/sparc
/opt/java/bin
/opt/SUNWspro/bin
/opt/SUNWspro/bin
/usr/dt/bin
/usr/ccs/bin
```

Selection: /utils/screenshots/ScreensDesignTools/

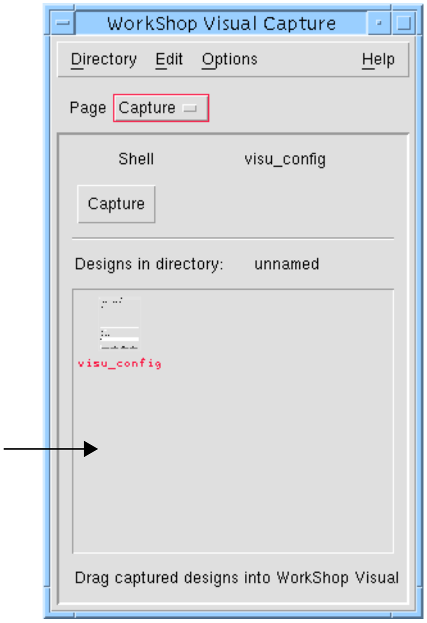
Extra scrolled list

FIGURE 13-7 Capture/Replay File Selection Box

You can also run the dialog from the command line. This is explained in “Sun WorkShop Visual Capture” on page 690.

Capture Dialog

The Sun WorkShop Visual Capture dialog is shown in Figure 13-8. It has two pages - one for Sun WorkShop Visual Capture and one for Sun WorkShop Visual Replay. You can change between pages by selecting from the option menu labelled “Page”. For details on the dialog when you are using Sun WorkShop Visual Replay see Chapter 14 ”Sun WorkShop Visual Replay“.



Window holding area containing thumbnail sketches of captured dialog designs

FIGURE 13-8 Capture Dialog

The text to the right of “Shell” displays the name of the currently active Shell of the associated application. This is the Shell which will be captured when you press the “Capture” button. Beneath the “Capture” button is an area containing the captured Shells in the current directory.

Saving and Accessing Captured Shells

By default, when applications are captured, files containing the captured design are created in a temporary unnamed directory.

Note – The temporary directory (and its contents) is removed automatically when you exit the application. If you want to preserve your work, you should be working in a named directory (see below).

Use the “Directory” menu to create a new directory, open an existing one or save the current directory under a different name.

When you open a directory, the dialog displays any captured designs that are contained there. Captured designs are displayed as “thumbnail” sketches of the Shell which was captured.

Using the operations in the “Edit” menu, designs can be cut or copied from one directory and pasted into another. The “Clear” command deletes the selected design.

By convention, captured designs are given the filename suffix “.xd”, although this suffix is not displayed in the Sun WorkShop Visual Capture dialog.

Note – A corresponding “.xpm” file is created with each captured dialog design. This is to enable Sun WorkShop Visual Capture to display the “thumbnail” sketch of the dialog in the window holding area.

Using Sun WorkShop Visual Capture

This section is a set of step-by-step instructions showing you how to use Sun WorkShop Visual Capture to capture the design of the user-widget configuration utility supplied with Sun WorkShop Visual, visu_config.

1. **Select “Sun WorkShop Visual Capture” from the “Tools” menu.**
2. **Type visu_config into the Command field of the Command Execution Dialog and press “Ok”.**

   The Sun WorkShop Visual Capture tool searches your path (the list of directories set by the PATH environment variable) for the named application, and invokes the first one it finds. Both the visu_config and Capture dialogs are displayed.

   Note that the Capture dialog is effectively attached to the visu_config program, so when you exit the visu_config application, Sun WorkShop Visual Capture also exits.
3. **Press “Capture” in the Capture dialog.**

   A thumbnail sketch of the active Shell of visu_config appears in the Capture dialog. To capture other dialogs, simply display them in visu_config and then press “Capture” again. Each dialog is added to the window holding area in the Sun WorkShop Visual Capture dialog.
4. **To see the design in Sun WorkShop Visual, drag (using mouse button 2) the thumbnail from the window holding area of the Capture dialog into Sun WorkShop Visual’s construction area.**

   As you do this, the Shell and its hierarchy appear in Sun WorkShop Visual.

Your Sun WorkShop Visual widget hierarchy now contains the widget structure of the initial visu_config dialog along with all associated resources. The dynamic display looks like visu_config. If you captured any sub-dialogs, they can also be dragged into the Sun WorkShop Visual construction area. They are added as separate dialogs.

Application Modal Dialogs

If your application runs an Application Modal dialog, you will not have access to the Sun WorkShop Visual Capture interface and therefore cannot use the Capture button. Instead, you must press the “hot key”. By default, this is set to the F5 function key in the Sun WorkShop Visual resource file. The hot key is translated to the function which effects the capture. The resource file entry is shown below:

```
*xdsTranslations:"<Key>F5: vcrInteractiveCaptureShell()"
```

You can change this to another key by editing the Sun WorkShop Visual resource file (which changes it for every user) or by editing the .Xdefaults file in your home directory (which changes it just for you). We have created a second “hot key” to ensure there are no clashes with your application, this can be accessed by the F12 key.

Note – If you would like more information on translations, you are advised to consult Chapter 4.3.2 in Volume Four of the “*X Toolkit Intrinsics Programming Manual*” published by O’Reilly and Associates, or any other comparable book.

Captured Information

You can see straight away that Sun WorkShop Visual Capture has captured the appearance of visu_config. In fact, more than that has been captured. Here is a list of what you have:

- The hierarchy of widgets as used in the original application
- All explicitly set resources.
- All layout including Form attachments
- Dimensions

Sun WorkShop Visual Capture gives you everything you need for the design of a Motif application. Callbacks, links and other *dynamic* actions associated with the application are not captured.

Capturing the Java Emulation Widgets

If you wish to use Sun WorkShop Visual Capture on a design which contains the new Java layout emulation widgets, you must check that the resource **xdsCaptureUserWidgets** is set to “true”. This resource ensures that non-Motif widgets are captured fully. If this resource is not set, the widgets appear as drawing areas in the capture script. For more information on the Java emulation widgets see “New Widgets for Java Classes” on page 323 in Chapter 10 ”Designing for Java“.

Capturing User Defined Widgets

You can capture designs containing third party widgets if you first set the resource **xdsCaptureUserWidgets** to “true”, as described above in “Capturing the Java Emulation Widgets” on page 430. You will be able to view these widgets in Sun WorkShop Visual only if they have already been integrated. For more information on integrating third party widgets see Chapter 23, “User-Defined Widgets”, starting on page 627.

Using Sun WorkShop Visual Capture From the Command Line

Sun WorkShop Visual Capture is supplied as a stand-alone application called `visu_capture`. Type: `visu_capture -x` to display basic information about the tool.

The following line gives an example of how Sun WorkShop Visual Capture can be used:

```
visu_capture -f MyCaptureDesign.xd AnApplication
```

`MyCaptureDesign.xd` is the name of the file which will contain the captured application. This file will use Sun WorkShop Visual’s save file format. `AnApplication` is the name of the application you wish to capture. Pressing the “hot key” (F5 by default) performs the capture.

See “Sun WorkShop Visual Replay and Sun WorkShop Visual Capture” on page 807 for tips and hints about using Sun WorkShop Visual Capture.

CHAPTER **14**

Sun WorkShop Visual Replay

Introduction

Sun WorkShop Visual Replay can record and playback any Xt based application.

In *record* mode, Sun WorkShop Visual Replay creates a script containing a high level description of the user's actions e.g. “push hello_button, type Hello World”.

In *playback* mode, you can check the state of any widget in the application and control the rate of playback. The actions in the script are replayed exactly as if the user were sitting at the keyboard.

Sun WorkShop Visual Replay has a user extensible command set which is powerful, easy-to-use and very flexible. It can be deployed in many ways:

- Record and playback Motif applications
- Produce demonstrators of your product
- Debug your application
- Recreate problems found in an application
- Develop tutorials for your product
- Automate the testing of an application

No recompilation or relink is necessary and no special test environment is required.

Sun WorkShop Visual Replay is available from the Sun WorkShop Visual “Tools” menu. You can also use the tool from the command line as described in “Recording and Replaying From the Command Line” on page 447.

This chapter starts by providing you with a description of the use of Sun WorkShop Visual Replay together with some simple tutorial examples to help you become acquainted with its use. “Extending the Sun WorkShop Visual Replay Widget Set” on page 458 and “Adding Your Own Sun WorkShop Visual Replay Commands” on page 471 describe how to extend the capabilities of Sun WorkShop Visual Replay.

Appendix A, “Sun WorkShop Visual Replay Command Syntax”, starting on page 811 provides detailed descriptions of the syntax of Sun WorkShop Visual Replay scripts.

Recording and Replaying Java Applications

Sun WorkShop Visual Replay can be used for Java applications. To do this, you should specify the Java interpreter as the first application and your target application afterwards, as explained in “Debugging With Sun WorkShop Visual Replay” on page 457. That section describes the use of *indirections* when Sun WorkShop Visual Replay is used from the command line. Thus, to record or replay a Java application, you would have to specify the Java interpreter too:

```
visu_replay java MyJavaProgram
```

This applies to Java applications but not to applets. For more information on Java code generation, see Chapter 10 ”Designing for Java“.

Before Using Sun WorkShop Visual Replay

For successful operation of Sun WorkShop Visual Replay, the Motif application you wish to record must have been *dynamically linked* with the Xt library (libXt). On many UNIX implementations, you can find out whether the application has been dynamically or statically linked with libXt by typing:

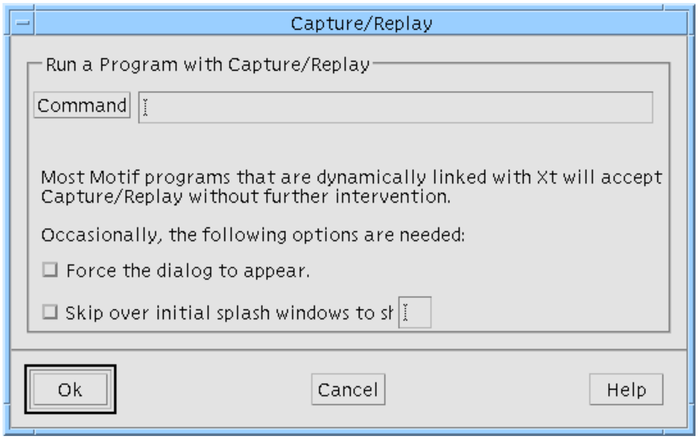
```
ldd AnApplication
```

If the output mentions libXt, the application has been dynamically linked with the Xt library and can be used with Sun WorkShop Visual Replay. If this library is not present, the application has probably been statically linked with the Xt library. You will have to relink your application with the Xt shared library if you want to use Sun WorkShop Visual Replay.

How to Invoke an Application with Sun WorkShop Visual Replay

Note – If you are keen to get started straight away with Sun WorkShop Visual Replay, you may wish to skip this section and move directly to “Tutorial” on page 443.

When you select “Sun WorkShop Visual Replay” from the “Tools” menu, a dialog is displayed requesting the name of the application you wish to record or replay. This dialog is shown in Figure 14-1.



Most Motif programs that are dynamically linked with Xt will accept Capture/Replay without further intervention.

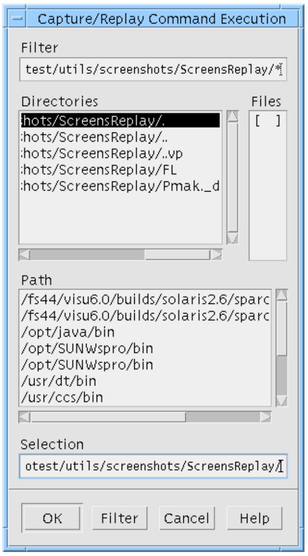
Occasionally, the following options are needed:

□ Force the dialog to appear.

□ Skip over initial splash windows to sh

FIGURE 14-1 Capture/Replay Application Prompt

Enter the name of the application in the text box labelled “Executable”. If you are unsure of the application’s name, or where to find it on your system, press the button labelled “Executable”. This produces a file selection box containing an extra scrolled list, as shown in Figure 14-2.



test/utils/screenshots/ScreensReplay/*

```
:hots/ScreensReplay/.
:hots/ScreensReplay/..
:hots/ScreensReplay/..vp
:hots/ScreensReplay/FL
:hots/ScreensReplay/Pmak._d
```

[ ]

```
/fs44/visu6.0/builds/solaris2.6/sparc
/fs44/visu6.0/builds/solaris2.6/sparc
/opt/java/bin
/opt/SUNWspro/bin
/opt/SUNWspro/bin
/usr/dt/bin
/usr/ccs/bin
```

otest/utils/screenshots/ScreensReplay/

Extra scrolled list

FIGURE 14-2 Capture/Replay File Selection Box

Each item in the extra scrolled list labeled “Path” is a directory from your PATH. Selecting an item from this list displays the contents of that directory in the “Files” list.

Note – The extra scrolled list uses the PATH set up for you when you ran Sun WorkShop Visual and may contain some extra directories required by Sun WorkShop Visual. When you exit Sun WorkShop Visual your PATH is the same as it was before running the application.

When you select an entry from the “Files” list and “OK” the dialog, the entry is placed in the “Executable” field of the Capture/Replay Application Prompt dialog. Enter any flags or arguments for the application in the text box labelled “Arguments” in this dialog. When you press “OK”, the application is run with Sun WorkShop Visual Replay.

Two points need to be made here:

1. The Sun WorkShop Visual Replay dialog is *part* of the application and will be dismissed when you exit from the application.

2. Sun WorkShop Visual Replay only *monitors* what happens in the application - it does not and cannot affect the operation of the application other than by simulating operations in playback mode.

What Gets Recorded

Sun WorkShop Visual Replay has been designed as an efficient way of exercising Motif interfaces with the emphasis on portability and clarity of description.

Sun WorkShop Visual Replay focuses on recording navigation between widgets within an application and the user interaction with those widgets. The following information can be recorded and replayed:

- actions on all widgets
- displaying/closing sub-dialogs.
- use of keyboard accelerators, modifiers.
- all keyboard input (text, arrow keys, delete etc.).
- all mouse button actions - including *which* mouse button has been pressed.

Sun WorkShop Visual Replay has not been designed as a general-purpose X testing engine and, consequently, there are some aspects of the use of an application which Sun WorkShop Visual Replay does not record. However, provision is given for you to extend the capabilities of Sun WorkShop Visual Replay. This is discussed in "Extending the Sun WorkShop Visual Replay Widget Set" on page 458 and "Adding Your Own Sun WorkShop Visual Replay Commands" on page 471.

The Sun WorkShop Visual Replay Interface

The Sun WorkShop Visual Replay dialog appears to the side of your application. A copyright message is also shown on standard error when Sun WorkShop Visual Replay starts up.

If the copyright message does not appear, your application has probably not been dynamically linked with the Xt library (see "Before Using Sun WorkShop Visual Replay" on page 434).

The Sun WorkShop Visual Replay dialog is shown in Figure 14-3.

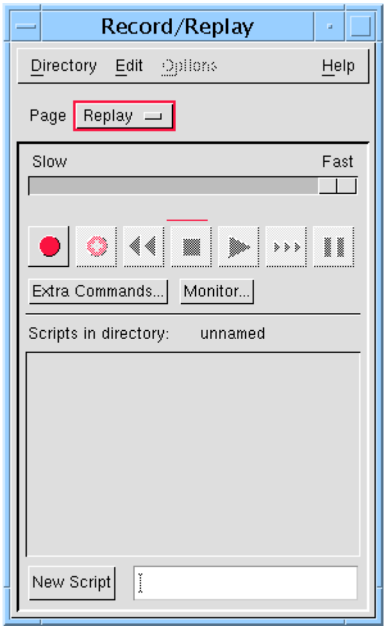


FIGURE 14-3 Sun WorkShop Visual Replay Dialog

You can also display this dialog when running Sun WorkShop Visual Replay from the command line. This is explained in “Recording and Replaying From the Command Line” on page 447.

This dialog has two pages - one for Sun WorkShop Visual Replay (*Replay*) and one for Sun WorkShop Visual Capture (*Capture*). You can change between pages by selecting from the option menu labelled “Page”. For details on the dialog when you are using Sun WorkShop Visual Capture see “Sun WorkShop Visual Capture” on page 425.

Functions and Operations

Once the Sun WorkShop Visual Replay dialog is displayed, you can begin to record / replay scripts straight away. All record / replay actions take place using the Sun WorkShop Visual Replay button panel which is shown in Figure 14-4.

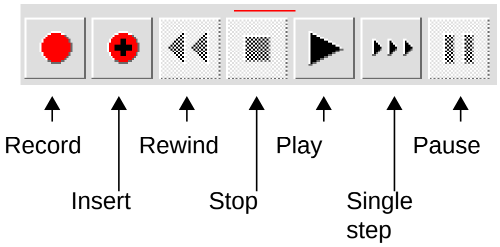


FIGURE 14-4 Sun WorkShop Visual Replay Button Panel

The buttons are described below:

● **Record** records user actions in the application from the current position in the selected script. If record is pressed after stopping a script, it will overwrite the script from that point on. Pressing record at the end of a script will append to it.

⊕ **Insert** records user actions in the application at the point where the script was stopped. Subsequent actions in the script are preserved.

◀◀ **Rewind** rewinds the selected script to the beginning.

Note – *To replay the script exactly, you may have to reset the application to the state from which the recording was started.*

■ **Stop** stops the playback of a script.

▶ **Play** plays the selected script from the current position in the script until either the script is stopped or reaches its end.

▸▸▸ **Single step** plays the next command in a script.

❚❚ **Pause** pauses a record or playback. Press the button again to continue.

Only valid buttons can be selected; all other buttons are grayed out.

Before you have created any scripts, the only button you can press is “Record”. This creates an “unnamed” script. Once you have created a script, you can “Rewind”, “Play” and “Single step” it.

The “Insert” button becomes active when the script is stopped or paused. The insertion process is described more fully in “Inserting in a Script” on page 447.

Creating and Naming Scripts

Press the “New Script” button to create an empty script. To name or rename a script, do the following:

1. Click on the associated icon in the Sun WorkShop Visual Replay dialog

2. Enter the name in the New Script text field and press Return

If you enter the same name as that of an existing script, a number is appended to the newly named script to differentiate between it and the original.

Note – If you have no scripts in the Sun WorkShop Visual Replay dialog, pressing “Record” will create a new “unnamed” script automatically.

Selection and Status Indicators

The currently selected script is highlighted in the Sun WorkShop Visual Replay dialog.

The Sun WorkShop Visual Replay *status indicator* shows you whether you are recording or replaying and where in the script you are. If the status indicator is red, it indicates that you are recording. Otherwise you are replaying. Figure 14-5 shows the possible states of the indicator:

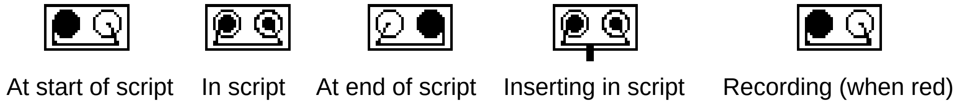
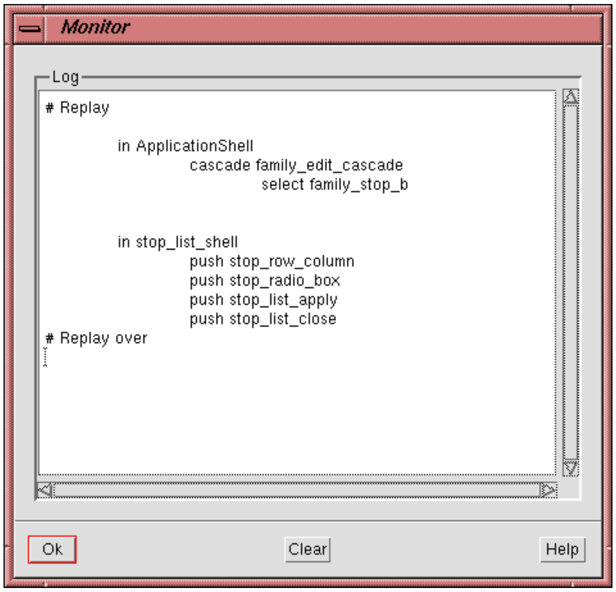


FIGURE 14-5 Sun WorkShop Visual Replay Indicator States

The last button you selected has a red line above it in the button panel.

Monitoring

The “Monitor” button displays a log of the actions you are taking while recording and replaying. Comments indicating the start and end of a record or replay session are inserted automatically by Sun WorkShop Visual Replay as demonstrated in Figure 14-6.



```
# Replay

          in ApplicationShell
                    cascade family_edit_cascade
                              select family_stop_b


          in stop_list_shell
                    push stop_row_column
                    push stop_radio_box
                    push stop_list_apply
                    push stop_list_close
# Replay over
```

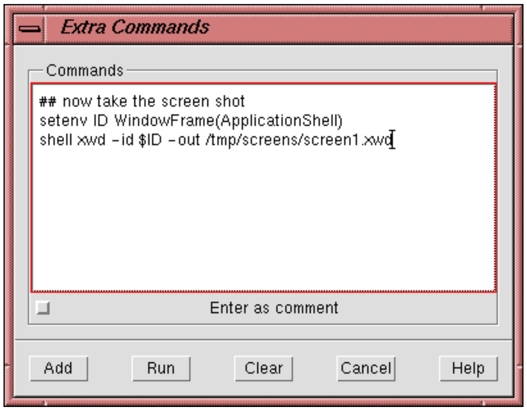
FIGURE 14-6 Sun WorkShop Visual Replay Monitor Window

Inserting Extra Commands

As well as actions, you can also add non-application commands and comments to a script. This can be done by editing the script by hand or via the Sun WorkShop Visual Replay interface. This section describes how to edit the script from the interface.

First stop the script at the point where the additional commands are to be placed. To place extra commands at the start of the script, you must first rewind it. To place commands at another point in the script, single-step to that point.

Next press the button labelled “Extra Commands”. This displays a text edit window into which the extra commands or comments can be entered. This dialog is shown in Figure 14-7.



```
## now take the screen shot
setenv ID WindowFrame(ApplicationShell)
shell xwd -id $ID -out /tmp/screens/screen1.xwd
```

FIGURE 14-7 Extra Commands Dialog

If the “Enter as comment” toggle is set, the contents of the dialog are treated as comments. Each line is prepended with a ‘#’ character in the script.

The “Run” button executes the commands in the dialog independently of the recorded script. Use the Monitor window to see the commands being executed. Once you are satisfied with the commands, press the “Add” button to store them in the script.

Press “Clear” before entering additional commands or comments. This removes the information from the Extra Commands dialog - it has no effect on the contents of the script.

Changing Replay Speed

The fast/slow slider on the Sun WorkShop Visual Replay dialog allows you to change the speed at which the selected script is replayed. By default, the script is played at the maximum speed.

Application Modal Dialogs

If your application runs an Application Modal dialog, you will not have access to the Sun WorkShop Visual Replay interface until you have closed the dialog. This means that you cannot stop recording or replaying within the dialog. In single-step mode, all actions within an Application Modal dialog are treated as a single step.

Saving and Accessing Scripts

By default, the scripts you create in the Sun WorkShop Visual Replay dialog are stored in a temporary unnamed directory.

Note – Unless the environment variable *XDS_KEEPDIR* is defined, the temporary directory (and its contents) is removed automatically when you exit the application. If *XDS_KEEPDIR* is set, the temporary directory and its contents are stored in */tmp/XDS_SAVE*. If you want to preserve your work, you should be working in a named directory (see below).

Use the “Save As” option from the Sun WorkShop Visual Replay Directory menu to save the current directory under a new name.

Use the “Open” option from the Directory menu to access scripts from another directory. The “Save As” option can also be used to rename the currently opened directory.

Using the operations in the “Edit” menu, scripts can be cut or copied from one directory and pasted into another. The “Clear” command deletes the selected script.

By convention, record scripts are given the filename suffix “.xds” in the file system. Note however that this suffix is not used to label the scripts in the Sun WorkShop Visual Replay dialog.

Tutorial

This section is a set of step-by-step instructions which demonstrates how to use Sun WorkShop Visual Replay to record interaction with the visu_config tool and then replay those actions.

Note – The tutorial requires no knowledge of visu_config. If, however, you would like more information on this tool, refer to “visu_config - the Main Dialog” on page 636.

---

1. **Select “Sun WorkShop Visual Replay” from the “Tools” menu.**
2. **Type: visu_config into the Executable field of the Capture/Replay dialog and press “Ok”.**

   This runs visu_config and displays the Sun WorkShop Visual Replay dialog alongside it.

3. **Press the “New Script” button.**

   This creates an “unnamed” script.

4. **Enter a name for the script in the New Script text field, followed by a carriage return.**

   The name of the script is changed accordingly.

5. **Press the “Monitor” button.**

   This brings up a dialog showing a log of all the actions for the session.

6. **Press the Record button, as shown in Figure 14-8.**

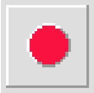


FIGURE 14-8 Record Button

7. **In visu_config, perform the following actions:**

   a. **Enter:** `one` **in the Selection text field and press Return.**

   The name is added to the “Families” list.

   b. **Double click over the name “one” in the Selection field, type:** `two` **and press Return.**

   The Families list now contains two entries.

   c. **Click on** `one` **in the Families list and press the “Edit” button.**

   The “Widget Classes” dialog is displayed.

   d. **Enter:** `WidgetOne` **in the Selection text field and press Return.**

   The name is added to the “Widget classes” list.

   e. **Double-click “WidgetOne” in the Widget classes list.**

   This displays the Widget dialog.

   f. **Press the “Close” button in the Widget dialog.**

   g. **Press the “Close” button in the Widget Classes dialog.**

   h. **Select the “Stop list” option from the “Edit” menu in the Families dialog.**

   This displays the Stop list dialog.

   i. **Press the toggles labelled “Pulldown Menu” and “Text Field” in the Stop list dialog.**

   j. **Press the “Apply” button followed by the “Close” button.**

k. **Select “New” from the File menu in the Families dialog.**

   The “Save changes” warning dialog is displayed.

l. **Press the “No” button in the Save changes dialog.**

8. **Press the “Stop” button**

   The “Record” and “Rewind” buttons become sensitive. All the other buttons become insensitive.

   A file has been created containing a record of your actions. This file can be replayed at any time. For the purposes of this tutorial, we are going to play it back straight away.

9. **Press the “Rewind” button.**

   The record, insert, play and single step buttons become sensitive.

10. **Press the “Play” button.**

    You can now see what you have recorded. Using the fast/slow slider in the Sun WorkShop Visual Replay dialog, you can change the rate at which your session plays back.

11. **Press the “Rewind” button.**

12. **Press the “Single step” button.**

    Using this button you can single step through each command in the record script. This is more informative if you have the Monitor window on the screen. As each step is replayed it is printed in the Monitor window.

13. **Exit visu_config.**

    Select “No” when you are asked if you wish to save the changes. The record session ends when the application exits. The Sun WorkShop Visual Replay dialog is also dismissed. This is because the dialog is, in effect, part of the visu_config program.

Note – Unless the environment variable *XDS_KEEPDIR* is defined, the temporary directory (and its contents) is removed automatically when you exit the application. If *XDS_KEEPDIR* is set, the temporary directory and its contents are stored in */tmp/XDS_SAVE*. If you want to preserve your work, you should be working in a named directory (see below).

The Contents of the Script

The example above produces the following script, annotated to show the steps in the tutorial. This script introduces some 90% of the Sun WorkShop Visual Replay syntax.

(in the visu_config application)

```
in ApplicationShell
```

Entering “one” in the Selection text field.

```
     push Text
     type one
     key Return
```

Entering “two” in the Selection text field.

```
     doubleclick Text
     type two
     key Return
```

Selecting “one” from the Families list.

```
     push ItemsList(‘one’,1)
     push family_selection.OK
```

Entering “WidgetOne” in the Selection text field in the Widget Classes dialog.

```
in entity_dialog
     push Text#5
     type WidgetOne
```

Selecting “WidgetOne” from the Widget Classes list.

```
     push widgetlist_selection.OK
     doubleclick ItemsList#5(‘WidgetOne’,1)
in widgetedit_dialog
```

Closing the Widget and Widget Classes dialog.

```
     push widgetedit_closeb
in entity_dialog
     push widgetlist_selection.widgetlist_quitb
in ApplicationShell
```

Displaying the Stop list dialog.

```
     cascade family_editb
             select family_stop_b
in stop_list_shell
```

Setting toggles in the Stop list dialog.

```
     push stop_pulldown_menu
     push stop_text_field
     push stop_apply
     push stop_close
```

Selecting New from the File menu.

```
in ApplicationShell
     cascade family_fileb
             select family_newb
```

Saying No to the Save changes prompt.

```
in savechanges_dialog
     push savechanges_messagebox.Cancel
```

Note – The file you created may not be exactly the same as this one because you may have performed the actions in a slightly different order or you may have made mistakes and gone back to correct them. All of this is recorded.

Inserting in a Script

You can insert at the beginning of a script or partway through it (i.e. during a single step sequence).

Note – You can find out exactly where you are in the script if you have the "Monitor" window open.

To add to a script:

- If you are at the beginning of a script, press the "Insert" button.
- If you are partway through a script, press the "Stop" button and then the "Insert" button.

In both cases, then continue using the application.

Pressing "Insert" is the same as pressing "Record" except that whatever you do in the application is *inserted* into the existing script at the current point. When not in Insert mode, pressing "Record" will overwrite whatever was in the script.

Note – Remember when inserting actions into a script that script must be able to continue after the insertion. If this cannot be done, the replay will stop at that point.

Recording and Replaying From the Command Line

Sun WorkShop Visual Replay is supplied as a stand-alone application which can be run from the command line both for recording and replaying scripts.

Using Sun WorkShop Visual Replay to Record Scripts

Sun WorkShop Visual Replay (when used to record user actions) is supplied as a stand-alone application called visu_record.

Type: visu_record `-x` to display basic information about the tool.

The following line shows how to use visu_record:

```
visu_record -f MyRecordScript AnApplication
```

`MyRecordScript` is the name of a file into which a script recording the session will be saved. You do not have to supply this parameter. If you do not, the script is written to standard output. `AnApplication` is the name of the application you wish to record. The `-i` flag tells Sun WorkShop Visual Replay that you wish to use the tool interactively. In this case, the Sun WorkShop Visual Replay dialog is displayed as described in "The Sun WorkShop Visual Replay Interface" on page 437.

Using Sun WorkShop Visual Replay to Play Back Scripts

Sun WorkShop Visual Replay, when used to play back recorded scripts, is supplied as a stand-alone application called `visu_replay`.

Type: `visu_replay` -x to display basic information about the tool.

The following line shows how to use `visu_replay`:

```
visu_replay -f MyRecordScript AnApplication
```

`MyRecordScript` is the name of a file containing the script of the recorded session. You do not have to supply this parameter. If you do not, the script is read from standard input. `AnApplication` is the name of the application you wish to rerun. The `-i` flag informs Sun WorkShop Visual Replay that you wish to use the tool interactively via the Sun WorkShop Visual Replay dialog.

Getting the Most From Sun WorkShop Visual Replay

This section describes the uses to which Sun WorkShop Visual Replay can be put and discusses:

- preparing rolling demonstrations
- taking screen dumps
- testing
- debugging

This list is neither definitive nor exhaustive - it serves only to demonstrate the wide-ranging capabilities of Sun WorkShop Visual Replay.

Preparing Rolling Demonstrations

A script prepared using Sun WorkShop Visual Replay can be run in a "continuous loop" using a simple shell script, as shown below:

```
while (true)
   {
         visu_replay -f mydemo.xds myapplication
   }
   end
```

Note – When preparing such a rolling demonstration, always ensure that the last part of your script has commands which place your application in a state from which it can be re-run.

Taking Screen Dumps

Taking screen dumps of an application can be a tortuous process - particularly if the application is constantly subject to change. Sun WorkShop Visual Replay allows you to create screen dumping scripts which can be reused at any time. And because the screen dumping process is now automatic, the cost of producing them falls dramatically.

A screen dumping script consists of a set of actions to prepare the application for the screen shot followed by non-application commands which actually do the screen shot. In the example script fragment shown below, a screen dump of the *current_shell* dialog is taken:

```
in current_shell
          setenv ID WindowFrame(current_shell)
          shell xwd -id $ID -out /tmp/current_shell.xwd
```

The last two lines are extra, non-application, commands. The first sets the variable `ID` to the current shell window, including its window decorations. The second uses the `xwd` command to get a snapshot of the shell window and store it. Of course, you can substitute `xwd` with any other screen dumping command of your choice. The keywords used for setting variables are discussed in "Non-Application Operations" on page 825.

Testing

Sun WorkShop Visual Replay is a simple-to-use, portable, and powerful widget-based testing tool. It is intended to provide a testing solution across the whole range of platforms that are supported by Sun WorkShop Visual.

The Role of Widget-Based Testing

Most Motif/Xt programming involves reusing the Motif widgets, and using the X Toolkit. Sun WorkShop Visual Replay testing focuses on the Xt widget hierarchy, both for controlling a test sequence and for checking whether a test has succeeded.

It is important to note that you are not checking whether the widgets themselves are correct - only that user interaction with those widgets produces the desired results within your application.

Note – This testing technique and strategy is highly resistent to test “rot”. Your test results should be the same, whatever the size, shape or quality of the display being used. Tests will only need to be added or updated if the application itself changes. And of course these tests will soon detect any changes which have not been reported to the tester!

Not all testing can be automated in this way. There will always be a need to visually inspect an application to check whether it looks right or whether any graphics programming (e.g. in drawing areas) has worked. While there will always be a requirement for looking and thinking, the widget-based testing strategy ensures that you can focus your attention on those few parts of the application that need it.

The Approach to Testing

Experience has taught us that there are three graduated approaches to the production of a testing script:

- recording and replaying pre-recorded scripts
- splitting large tests into smaller fragments
- data-driven testing

Recording and Replaying Pre-Recorded Scripts

This is the simplest way of checking that user actions can be replayed exactly as they were recorded. However, the scripts can become very large and troublesome to maintain. It can also be difficult to work out which part of a test is failing. More importantly, any change to the application will mean that the whole script will need to be re-recorded.

Script Fragmentation

Here a large script is split into small, self-contained scripts each of which exercises an identifiable part of the application. Since this is such an effective testing technique, we have provided a detailed example in "Using Testing Macros" on page 455. Each fragment is expanded using a preprocessor (e.g `m4` or `cpp`), or any programming language you feel comfortable with. This allows you to build scripts such as:

StartApplication()

OpenFile(foo.c)

CloseApplication()

Your preprocessor, interpreter or compiler would then translate these fragments into a full Sun WorkShop Visual Replay command sequence.

This simple strategy takes you away from "step-by-step" programming, and your test scripts will be far more manageable.

The language you use for expressing your tests should be carefully selected. The main criteria should be:

- Ease of expression. If it is hard to describe your interface using the language, then your model will be harder to write, harder to read and harder to understand.
- Familiarity. If you use a language you are at ease with, then you can concentrate on your model.

Class based languages such as `Java` or `Python` are ideal for this purpose. Modelling languages, tailored for symbolic processing, such as `Lisp` or `Prolog` are other obvious candidates. Preprocessors such as `m4` or even the C preprocessor will get you going very quickly.

Alternatively you may prefer to build your model in the language used by your application. In this way you guarantee that it is always available when you port your software. The only rule of thumb is that if you feel you're writing a program rather than designing a set of tests, there is almost certainly an easier way.

This testing method is appropriate for most small to medium-sized applications. However, for very large applications (and Sun WorkShop Visual is a good example) fragmentation also has its limitations:

- You can easily end up with a large number of fragments
- Many fragments will be doing similar things in different dialogs - you might have an open fragment for each dialog in an application
- The fragments are procedural and prescriptive. While a fragment may contain lots of useful information about a dialog, you can only use it in the way it was intended.

The next sub-section describes how to overcome these problems.

Data-Driven Testing

Our experience in devising tests for Sun WorkShop Visual has shown that the most cost-effective way of writing tests is to provide a description of each dialog and then use that description in the tests. Consider the following example where Sun WorkShop Visual's Color Dialog is described:

```
ColorDialog.shell= my_color_shell
ColorDialog.helpbutton= color_help
ColorDialog.applybutton = color_apply
ColorDialog.quit        = color_quit
```

The names on the left provide an indirect way of referring to the widgets in a dialog. The names on the right are the specific widget names. Such descriptions could then be used in general purpose routines by simply passing in the name of the dialog, for example:

```
CheckHelpFor(ColorDialog)
Close(ColorDialog)
```

The definitions of `CheckHelpFor` and `Close` are shown below:

```
#define CheckHelpFor(dialog)
        in dialog.shell
               push dialog.helpbutton
  #enddef
#define Close(dialog)
        in dialog.shell
               push dialog.quit
  #enddef
```

These routines could be used for any dialog with a description such as that listed for ColorDialog, to check that help and close buttons have been provided.

This technique allows you to separate out the *description* of the interface from the *actions* which exercise it. It also means that any change to the interface requires only a change to the associated data description - test scripts remain unchanged. If a new dialog is introduced to the application, you simply have to write its description and any non-standard operations which may be performed on or in it.

The biggest advantage of such a strategy is that the description is simple, clear and so close to the design itself that keeping tests in sync with product development becomes a well defined and straightforward exercise.

Checking Test Success/Failure

A good test is one which has been designed to break that part of the application it is checking. The test is successful if the application does not fall over, otherwise it is a failure.

Automated replay, by itself, is a minimal form of testing. If the sequence replays without error, then you have some measure that what was expected did actually happen. It is minimal because it only tests one potential result of a user action.

Consider the action of opening a file. In a minimal test, the expected result would be that the file is opened and everything progresses smoothly. However, this test is by no means complete. You need to consider other (potential) results, e.g.

- what happens if the file is not accessible
- should status indicators change when the file is loaded
- if the file is read-only, is a warning dialog displayed

Using Control Flow and Expressions in a Test Script

The simplest test is one which records a series of actions within your application and then replays the script to duplicate those actions. While successful execution of such a script can give some confidence in your application, you can gain even greater confidence by taking advantage of the extra commands for control flow and expressions provided by Sun WorkShop Visual Replay to enrich a basic script. These allow you to cater for different display types, check widget resource settings, print messages, and much more.

Consider the situation where your application displays a message when it is running on a monochrome display but displays no message when it is running on a full color display.

Clearly, you don't want to have a separate test for each display. Instead, you can insert commands at the point where you expect the message to appear and wrap these commands in an `if` statement, e.g.

```
if !IsPseudoColor
    message Non PseudoColor display
    in warning_popup
    push warning.OK
endif
```

This same check will work whatever display hardware or window manager you are using.

The size of application dialogs is also important. Two dialogs shown simultaneously may both be fully visible on one display, overlap on another or be placed one on top of the other on a third. This can result in application-modal warning messages disappearing behind the main dialog, and your application apparently locking-up.

The following test script fragment demonstrates how to handle such a problem:

```
if !IsVisible(open_file_dialog)
    error The Open File dialog is off screen
endif
```

See "Display Expressions" on page 829 for more information on handling different display types.

If your application exhibits different behavior on different displays, your tests need to be written to accommodate this. For example, the application may put up a warning dialog to tell the user to expect some degradation of display quality.

Now consider the selection of an option from an option menu. While a standard script will certainly make the selection, a good testing script will check that the selection has been made.

The example below shows how we test that the Language option has been set to an expected value in the Sun WorkShop Visual Generate dialog:

```
  if !languageOption->menuHistory:'cppButton'
          message FAIL: Language option error.
          printres languageOption->menuHistory
          message expected cppButton
    endif
```

What to Do When a Test Fails

There are three ways to deal with a test failure:

- stop the test but stay in the application
- stop the test and exit from the application
- abort the test and carry on with the next test in the test sequence

Each relates to a particular `visu_replay` command line flag:

- `-user-on-error` - stays in the application
- `-exit-on-error` - exits from the application
- `-skip-on-error` - skips to the next test

The best way to handle failure is to prepare for it in your script. Use conditional sequences and take appropriate actions (e.g. output a message) when a failure occurs.

Another useful aid to the location of test failure is the `-v` command line flag. This displays commands from the script on standard out as they are executed. Once you have located the problem, you can create a smaller script to reproduce it. This can then be used (perhaps in conjunction with your favorite debugger) to identify the problem. It can also be added to your regression test suite to demonstrate that the bug has been fixed.

Using Testing Macros

We described in "Script Fragmentation" on page 451 how test scripts can be modularized using macros which define actions which are repeated (e.g. opening dialogs, starting the application, typing into a text field etc.) This makes the scripts easier to create, amend and check by hand.

Example of Scripts Using Macros

In order to illustrate how macros can be used to create modular scripts, an extract from the Sun WorkShop Visual test scripts is listed below as an example. This short script does the following:

1. Starts Sun WorkShop Visual
2. Creates a design containing a Shell and Form
3. Gives the Form a variable name
4. Saves the design, specifying a filename
5. Exits Sun WorkShop Visual

To do this in a way which makes the top-level script more readable, we shall use the macro preprocessor, m4. This is available on all UNIX systems.

The high-level script to do the above is:

```
include(Defs.m4)
StartUp()
shell   date
Palette(xd_XmDialogShell)
Palette(xd_XmForm)
VariableName(myform)
SaveDesignAs(mydesign.xd)
message Test Sequence Over
shell   date
Finish()
```

Most of the above script consists of macro calls. See Appendix A, “Sun WorkShop Visual Replay Command Syntax”, starting on page 811 for more details on which part of the syntax are keywords.

The macro definition script, named *Defs.m4,* looks like this:

```
define(HandleExpectedWarning,
        in warning_popup
                push warning.OK)
define(StartUp,
        if !IsPseudoColor
                message Non PseudoColor display
                HandleExpectedWarning()
        endif)
define(Palette,
        in ApplicationShell
                push $1)
define(VariableName,
        in ApplicationShell
                 multiclick nb_vn_t
                 type $1
                 key Return)
define(SaveDesignAs,
        in ApplicationShell
```

```
                cascade file_menu
                        select fm_menu.fm_saveas
                in save_dialog_popup
                        doubleclick Text
                type $1
                 push save_dialog.OK)
define(Finish,
        in ApplicationShell
                cascade file_menu
                                select fm_menu.fm_exit
                if in save_changes_dialog
                        push xd_question.xd_question_cancel_b
                endif)
```

The following command:

```
m4 Test.in > Test.xds
```

creates the final script file which can be passed to Sun WorkShop Visual Replay. The file *Test.in* is the high-level script and *Test.xds* is the output file which will contain the final script with expanded macros.

Try out this example by typing in the files listed above and then using m4 to make the final script file. Having done this, run Sun WorkShop Visual Replay with Sun WorkShop Visual specifying *Test.xds* as the script to be replayed:

```
visu_replay -f Test.xds visu
```

You could take this example one step further by defining the names of the widgets on the Sun WorkShop Visual widget palette in a separate file and then defining the "Palette" macro so that it looks up the widget name from a high-level name such as "shell" or "form":

```
define(shell, xd_XmDialogShell)
define(form, xd_XmForm)
```

In this way the internal names are kept in one place where they can be maintained and changed more easily.

Debugging With Sun WorkShop Visual Replay

Running Sun WorkShop Visual Replay from the command line allows you to provide more than one application name if the application is an *indirection*.

For example, the following command:

```
visu_replay -f MyScript dbx AnApplication
```

would run a dbx session on the application AnApplication. Any debugger can be used. Using Sun WorkShop Visual Replay means that you can reach the stage at which you wish to start debugging quickly. To break into the debugger you can either reach the end of the script or place a “breakpoint” in the script. “breakpoint” is a keyword which is followed by the name of a widget. When the widget is activated, the application breaks into a debugger.

Extending the Sun WorkShop Visual Replay Widget Set

Overview

Sun WorkShop Visual Replay is based on the principle that the actions which are recorded in a script must be immediately recognizable as *user actions*.

Most actions which take place within a Motif application are described in terms of how a user interacts with its widgets (e.g. by clicking with one of the mouse buttons) or what is typed from the keyboard. This makes recording and replaying a Motif application very straightforward. It also makes it easy for a tester to understand, program and maintain scripts. The same must be true of any non-standard widget used in an application.

There are a number of Motif widgets (those for which the position in the widget is important) for which this approach does not immediately work. For most, e.g. Scales, ScrollBars, etc., a single mechanism will work for all instances of that widget. In a DrawingArea, or other custom widget, each instance of a widget may behave quite differently.

For example, although the recording software may observe a click in a drawing area, the user sees this action quite differently. He is interacting with objects that have been drawn in the drawing area by the application. But these objects only appear within the code of the application - they are not part of the interface.

Since you, as an application programmer, will know exactly what a click in a particular custom widget or drawing area actually means, you can easily provide routines which describe these actions so that they can be understood and used by people who wish to record and replay your widget.

If you are programming a drawing-area, or some other customized widget, you will already have written code to convert from an event at a particular (x,y) coordinate in that widget to a particular action in the application. Sun WorkShop Visual Replay provides interfaces which allow you to register converters that allow testers to make use of your routines.

You need to provide Sun WorkShop Visual Replay with two converters: one for recording, which converts an event at a particular (x,y) coordinate into an action and another, for replaying which converts that action to an (x,y) coordinate.

The conversion routine allows you to map the (x,y) coordinate to something which makes sense both to the user and to the widget itself.

Note – Some widgets provide an easy way to convert between (x,y) coordinates and the internal structure of the widget, e.g. the *XmListYToPos* function. This is the preferred method. Other widgets provide ways of determining and changing the state of a widget. For example you use *XmScrollBarGetValues* to record a user action on a scroll bar and *XmScrollBarSetValues* to replay that action. You can use the converters to program a widget directly if the event strategy is difficult to implement.

The same mechanism is used both for widget classes (e.g. third party widgets) and for custom widgets (e.g. the Motif XmDrawingArea widget).

The next two sections describe the converter routines. We then give an example which shows the creation of converters for the Motif XmList widget class.

Event to Name/Attribute Conversion Routine

Name

xdsXyToNameProc - interface definition for procedure used to convert from an event to a name/attribute description

Synopsis

```
typedef int (*xdsXYToNameProc) (
    Widget widget,
    int    x,
    int    y,
    char**  name_p,
    char** attribute_p )
```

Inputs

widget the widget that will use the routine

x the x co-ordinate of the event

y the y co-ordinate of the event

name_p (return) pointer to a string that identifies the part of the widget

attribute_p (return) pointer to a string that adds to the name, e.g. center, left, right

Usage

The routine should return 0 on failure, 1 on success. The strings that you assign to `name_p` and `attribute_p` are not freed by Sun WorkShop Visual Replay. Since copies are taken, you can use static storage.

If the routine fails, an error is reported.

Name/Attribute To Event Conversion Routine

Name

xdsNameToXYProc - interface definition for procedure used to convert from a name/attribute description to an event

Synopsis

```
typedef int (*xdsNameToXyProc) (
    Widget widget,
    char*  name,
    char* attribute,
    int*  x_p,
     int*  y_p )
```

Inputs

widget	the widget that will use the routine
name	a string that identifies the part of the widget
attribute	string that adds to the name, e.g. center, left, right
x_p	(return) pointer to the x co-ordinate result
y_p	(return) pointer to the y co-ordinate result

Usage

The routine should return 0 on failure, 1 on success.

Notes

Sometimes it is very easy to program the effect you need to replay directly onto the widget, e.g. by setting a resource value or calling a convenience function, but extremely difficult to mimic the event sequence precisely. In these circumstances, you can handle it yourself in the routine.

You should still return success, but set the (x,y) co-ordinates to negative values. The standard mechanism simulates a single click within a widget and expects positive coordinates.

An Example

This worked example comes from the Sun WorkShop Visual Replay sources. It is an example of how to register converters for a class of widgets, in this case the Motif XmList widget class. It demonstrates how a click in an XmList widget can be converted to the selection of a particular instance of an element from that list. This is the actual mechanism used for XmList widgets by Sun WorkShop Visual Replay. An example of its use is illustrated in the script fragment below:

```
  in my_shell
          push my_list_widget(‘this line’,1)
```

When the script is replayed, a button click is simulated at the appropriate (x,y) coordinates within the widget.

Once you have read through this example, you will be able to:

- integrate a 3rd party widget
- use the conversion mechanisms and understand the `widget(name,attribute)` notation
- understand how we have implemented the XmList widget class

The source files for the example, together with a Makefile are provided in the $VISUROOT*/src/examples/replay/cvtXm* directory, where $VISUROOT is the location of your Sun WorkShop Visual installation.

The contents of this directory are listed below:

File	Description
Makefile	allows you to build the shared object on different platforms
README	shows how to build your shared object
motif*.c	conversion routines
register.c	registration routines
xds*	support files

The support files provide the framework which allows your shared object to communicate with the Sun WorkShop Visual Replay engine. You do not need to change any of these files.

For the purposes of this example, we will be examining *motif2.c* which contains the XmList converters and *register.c* which includes code to register these converters.

The example illustrates the three stages in extending the Sun WorkShop Visual Replay widget set:

1. Event (x,y) to Name/Attribute pair - `xdsXYToNameProc`
2. Name/Attribute pair to Event (x,y) - `xdsNameToXYProc`
3. Registering Converters - `xdsRegisterContextHandler`

The three associated routines are:

1. `xdsListXyToName()` - which takes an (x,y) position and returns a name/attribute pair.
2. `xdsListNameToXy()` - which takes a name/attribute pair and returns an (x,y) position.
3. `xdsRegister()` - which registers a widget and its associated converters with Sun WorkShop Visual Replay

What is important is the structure of the converter code and how the converters are registered and not *how* we have implemented the XmList converters.

xdsListXyToName()

This function is in *motif2.c*. It converts an (x,y) coordinate to a name/attribute pair, illustrating the `xdsXyToNameProc` interface definition structure. This function is used when Sun WorkShop Visual Replay is in **record** mode.

```
int
xdsListXyToName( widget, x, y, namep, attrp)
    Widget widget;
    int x, y;
    char ** namep;
    char ** attrp;
{
    extern Boolean XmStringCompare();

    extern char *  xdsCvtXmStringToString();
    extern Boolean xdsCvtSetListError();
    extern int     xdsCvtListFailure();
    extern Boolean xdsCvtGetXmListEntries();

    extern int XmListYToPos();

    static char name[255];
    static char count[20];

    int pos;
    int len = 0;
    int n;

    int instance = 1;
    XmString * list = (XmString*)0;
    XmString   item;

    /* get the element */
    if (!xdsGetXmListEntries[1]( widget,&list, &len)) {
          return xdsListFailure();
    }

    /* use XmListYToPos() to get the list element */
    pos = XmListYToPos[2]( widget, (Position)y);
    if (pos < 0 || pos > len) {
          xdsCvtSetListError(LIST_OUT_OF_BOUNDS);
          return xdsCvtListFailure();
    }

    item = list[--pos];
    for (n = 0; n < pos; n++) {
          if (XmStringCompare( item, list[n]) == True)
                 instance++;
    }
    /* prepare the description */
    (void) sprintf ( count, "%d", instance);

    (void) strcpy  ( name, xdsXmStringToString[3](item));

    *namep  = name;
    *attrp  = count;

    return 1;
}
```

xdsListNameToXy()

This function is also in *motif2.c*. It converts a name/attribute pair to an (x,y) coordinate, illustrating the `xdsNameToXyProc` interface definition structure. It is the complementary function to `xdsListXyToName()`. This function is used when Sun WorkShop Visual Replay is in **replay** mode

[1] **`xdsGetXmListEntries()`** - we are going to return the element in the list; this is a simple routine that fetches the elements of an XmList.

[2] **`XmListYToPos()`** - the Motif convenience function `XmListYToPos( widget, y)` does all the conversion that we need. It takes the y-coordinate of the event, and returns its position in the list.

[3] **`xdsXmStringToString()`** - a routine to convert an XmString to a String.

```
int
xdsListNameToXy( widget, name, attr, xp, yp)
    Widget widget;
    char * name;
    char * attr;
    int  * xp;
    int * yp;
{
    extern char * xdsCvtXmStringToString();
    extern Boolean xdsCvtSetListError();
    extern int    xdsCvtListFailure();
    extern int    xdsCvtSetListItem();
    extern Boolean xdsCvtGetXmListEntries();

    Position x, y;
    Dimension w, h;
    int pos;
    int len = 0;
    int n;
    char * s;
    int instance = 1;
    XmString * list = (XmString*)0;
    XmString   item;

    if ((instance = atoi(attr)) == 0) {
         xdsCvtSetListError(LIST_BAD_INSTANCE);
         return xdsCvtListFailure();
    }
    instance--;

    if (!xdsCvtGetXmListEntries( widget, &list, &len)) {
         xdsCvtSetListError(LIST_EMPTY_LIST);
         return xdsCvtListFailure();
    }

    for ( n = 0; n < len; n++) {
          s = xdsCvtXmStringToString(list[n]);
          if (strcmp( name, s) != 0)
                 continue;
          if (instance--)
                 continue;
          break;
    }

    if (n == len) {
          xdsCvtSetListError(LIST_ELEMENT_NOT_FOUND);
          return xdsCvtListFailure();
    }
    (void) xdsCvtSetListItem( widget, n+1);

    if (!XmListPosToBounds[1]( widget, n+1, &x, &y, &w, &h)) {
          xdsCvtSetListError(LIST_OUT_OF_BOUNDS);
          return xdsCvtListFailure();
    }

    *xp = x + (w/2);
    *yp = y + (h/2);
    return 1;
}
```

xdsRegister()

The function is called in *register.c*. It registers the two converters in *motif2.c* and those for the XmScrollBar, XmScale and XmDrawingArea widgets.

```
void
RegisterWidgets()
{
    extern Boolean xdsRegister();
    extern int xdsListNameToXy();
    extern int xdsListXyToName();
```

[1] **XmListPosToBounds**() - the Motif convenience function, XmListPosToBounds(), gives us the window bounding-box of a particular item in the list. This can be used to work out likely (x,y) coordinates for a click on that element.

```
    extern int xdsScrollBarNameToXy();
    extern int xdsScrollBarXyToName();
    extern int xdsScaleNameToXy();
    extern int xdsScaleXyToName();
    extern int xdsDaNameToXy();
    extern int xdsDaXyToName();

    (void) xdsRegister( "XmList", xdsListNameToXy, xdsListXyToName);
    (void) xdsRegister( "XmScrollBar", xdsScrollBarNameToXy,
xdsScrollBarXyToName);
    (void) xdsRegister( "XmScale", xdsScaleNameToXy,
xdsScaleXyToName);
    (void) xdsRegister( "XmDrawingArea", xdsDaNameToXy,
xdsDaXyToName);
}
void RegisterThisListWidget(
    Widget w;
{
    xdsRegisterContextHandler(w, xdsListNameToXy, xdsListXyToName);
}
```

The function is defined in *xdsSetup.h* and illustrates the `xdsRegisterContextHandler` interface definition structure.

```
Boolean
xdsRegister( classname, name2xy, xy2name)
    char * classname;
    int_f  name2xy;
    int_f  xy2name;
{
    bool_f bf = xdsGetRegisterFunction();

    if (!bf)
         return False;

    return (*bf)( classname, name2xy, xy2name);
}
```

Building the Example

The supplied *Makefile* is configured to build a shared object. A number of operating systems are supported. These can be listed by typing: make.

You only need to change the OBJECT line in the Makefile in order to build the shared object. This should be changed to:

```
OBJECT = cvt<classname>
```

where classname is the prefix of the widget class. In this example, the widget class is XmList, so we use the Xm prefix, i.e.

```
OBJECT=cvtXm
```

To create the shared object, type: make <*system*>. For example on a Solaris machine, you would type: make solaris. This would create a shared object called *libcvtXm.so*.

Once the shared object has been built, copy or link it into the directory *$VISUROOT/lib/xds*. It will then be loaded by Sun WorkShop Visual Replay when required.

The source files for registering converters, together with a Makefile are provided in the $VISUROOT*/src/examples/replay/cvtTemplate* directory, where $VISUROOT is the location of your Sun WorkShop Visual installation.

Adding Converters for Customizable Widgets

The example described above relates to widget *classes*. For customizable widgets (i.e. a specific *instance* of a widget, such as a Motif XmDrawingArea) a mechanism for registering conversion routines is provided for you in the Sun WorkShop Visual distribution. This allows you to tailor the behavior of Sun WorkShop Visual Replay in order to allow it to record and replay user actions within individual instances of widgets (Motif or non-Motif).

The converter registration code is listed below:

```
int_f _xdsRegisterFunction = (int_f)0;
Boolean
xdsRegisterContextHandler( widget, name2xy, xy2name)
    Widget widget;
    int_f  name2xy;
    int_f  xy2name;
{
    if (!_xdsRegisterFunction)
```

```
          return False;
    return (*_xdsRegisterFunction)( widget, name2xy, xy2name, True);
}
```

A call must be made to this function in the application.

Note – The *_xdsRegisterFunction* function pointer variable is set to 0. This means that the routine will always return and do nothing in your application when it is run without Sun WorkShop Visual Replay. When you run *with* Sun WorkShop Visual Replay, the variable is set to point to the Register handler which then gets called.

The routine for registering converters is described below.

Registering Converters

Name

xdsRegisterContextHandler - interface definition for procedure used to register a converter

Synopsis

```
Boolean xdsRegisterContextHandler(
    Widget widget,
    xdsNameToXYProc name2xy,
    xdsXYToNameProc xy2name)
Boolean xdsRemoveContextHandler( Widget widget)
```

Inputs

widget	the widget that will use the routines
name2xy	pointer to a conversion function for replay
xy2name	pointer to a conversion function for record

Description

This routine is for registering your own interpretations of events in a widget. You can call `xdsRegisterContextHandler` at any time after you have created the widget in your code, for example:

```
button1 = XmCreatePushButton ( shell1, "button1", al, ac );
   xdsRegisterContextHandler(shell1, func1, func2)
```

When replaying, Sun WorkShop Visual Replay will call the func1 routine. When recording, it will call the func2 routine.

The mechanism is available to you either as a source file (*client.c*), or as a precompiled library module (*libxdsclient.a*). In the former case, it has to be compiled with your application, in the latter case re-linked with it.

It has no impact on the application itself and can be left in it with no adverse effects.

Summary

To extend the Sun WorkShop Visual Replay widget set:

- Create an (x,y) to name / attribute converter
- Create a name / attribute to (x,y) converter

For widget classes:

- Register the converters
- Create a shared object and copy or link it to the *$*VISUROOT*/lib/xds* directory

For individual widgets:

- Call `xdsRegisterContextHandler` to associate a widget with a specific function
- Build your application with *client.c* or link it with the *libxdsclient.a* library

Use the contents of the supplied *examples* directory as a guide to writing, registering and building your converters.

Adding Your Own Sun WorkShop Visual Replay Commands

Overview

The command set of Sun WorkShop Visual Replay is intended for replaying user actions and for checking the state of an application with respect to its widget hierarchy and its resource settings. You are not limited to this set of commands. You can extend it to include commands to meet your own needs, for example:

- To produce screen dumps at various points in a replay session.
- To do other sorts of consistency checking on the widget hierarchy - one example would be to interface with Doug Young's *widgetlint* library.
- To insert a probe or a patch for a particular debugging problem. This will be of most use in a stripped optimized binary, where you do not have access to the full power of the debugger.

You have already seen in the previous chapter examples of user-defined modules which are loaded *implicitly* by the Sun WorkShop Visual Replay engine. You have also seen how to construct and build these modules.

import allows you to load a module of your own commands *explicitly* into a script. Once the module has been loaded the commands in it can be invoked using the *user* command. This section shows you how to produce such a module. The process is similar in many ways to that described in the preceding section.

An Example

We will describe how to create a module which contains one command. This command prints a message on standard error and the name of the current shell widget. You can use this example as a template for constructing your own commands.

The source files for the command, together with a Makefile are provided in the $VISUROOT*/src/examples/replay/usertemplate* directory, where $VISUROOT is the location of your Sun WorkShop Visual installation.

The contents of this directory are listed below:

File	Description
Makefile	allows you to build the extra command module on different platforms
README	shows how to build your command module
interface.c	contains the code for the extra command described in this section
xds*	support files

The support files provide the framework which allows your extra commands to communicate with the Sun WorkShop Visual Replay engine. You only need to change the `xdsResources.h` file - **the remaining files prefixed with xds need not be altered in any way**.

You only need to change the `OBJECT` line in the Makefile in order to build the module. Then build the module by typing:

```
          make <systemname>
```

You then copy or link the shared object to the *$*VISUROOT*/lib/xds* directory:

The contents of the `interface.c` file are shown below:

```
#include <stdio.h>
#include <X11/Xos.h>
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>

void
exampleHalloWorld( shell, message)
    Widget shell;
    char * message;
{
    if (!message)
         message = "no message";
    (void) fprintf ( stderr, "Widget %s says '%s'\n",XtName(shell),
message);
}
```

As you can see, a user-defined function should have two arguments:

- **shell** - is the current shell widget (passed in by Sun WorkShop Visual Replay)

- **message** - a string

The message is all the text which follows the **user** *command* syntax on that line in the script. The example script fragment below shows how a command would be accessed and used:

```
import usertemplate
     in ApplicationShell
         user HalloWorld I’m here
```

Here the message is “I’m here”.

The Interface

The interface between all objects and the Sun WorkShop Visual Replay engine takes place using the standard Xt resource handling routines.

Note – If you would like more information on resource structures, you are advised to consult Chapter 10 in Volume Four of the “*X Toolkit Intrinsics Programming Manual*” published by O’Reilly and Associates, or any other comparable book.

An entry for the new command is added to the resource list in `xdsResources.h`, as shown below:

```
  {
          “HalloWorld”, XtCCallback, XtRPointer, sizeof(XtPointer),
XtOffsetOf(data_t,HalloWorld), XtRImmediate,
(XtPointer)exampleHalloWorld
  }
```

Only three items are of significance within this code:

- `“HalloWorld”` is the resource name
- `XtOffsetOf(data_t, HalloWorld)` gives the offset to the relevant entry in the data structure
- `(XtPointer)exampleHalloWorld` is a pointer to the address of the function

A pointer to that resource is added to the data structure within this file:

```
typedef struct {
     int        type;
     XtPointer setValues;
     XtPointer getValues;
     XtPointer engineSetValues;
```

```
    XtPointer engineGetValues;
    /*----------------------*/
    XtPointer HalloWorld;
} data_t;
```

Entries above the line in the data structure are common to all Sun WorkShop Visual Replay objects.

The last thing to do in this file is to declare the function:

```
extern void exampleHalloWorld();
```

Building the Module

As in the preceding chapter, you only need to change the `OBJECT` line in the Makefile in order to build the module. For this example, we change it to:

```
OBJECT=usertemplate
```

and then build the module by typing:

```
make solaris
```

Finally, we copy or link the shared object we have built to the $VISUROOT*/lib/xds* directory:

```
cp libusertemplate.so $VISUROOT/lib/xds
```

That is all there is to it.

Summary

To add a new command to the Sun WorkShop Visual Replay command set:

1. Add the associated function to the *interface.c* file.
2. Add an entry to the resource list in *xdsResources.h*
3. Add a function pointer to the data structure in *xdsResources.h*
4. Add an extern declaration of the function in *xdsResources.h*
5. If necessary, change the OBJECT line in the Makefile
6. Build the module
7. Copy or link it to the $VISUROOT*/lib/xds* directory

You can create as many modules as you wish and load them into a script at any time.

Allowing Your Applications to Be Recorded and Replayed

To permit users to use Sun WorkShop Visual Replay to record and replay your application, you must do the following:

- have the following line in your code:

  ```
  xdsAllowUserAccess()
  ```

- link the application with the *libxdsclient.a* library

See “Sun WorkShop Visual Replay and Sun WorkShop Visual Capture” on page 807 for tips and hints about using Sun WorkShop Visual Replay.

CHAPTER 15

Groups

Introduction

In Sun WorkShop Visual you can select one or more widgets and make them into a *Group*. A Group can be used as a shortcut means of referencing either large numbers of widgets or widgets which are performing a similar function. They are, however, the fundamental building block of Smart Code. Smart Code is the toolkit-independent layer of code which Sun WorkShop Visual can generate for you to help you move your Motif application to other platforms *and* make the most of Internet technology.

Since Smart Code uses Groups as its basic data structure, you need to understand how to create them, customize them and use them first. This chapter describes Groups. For more information on Smart Code, see:

1. Chapter 16, “Get/Set Smart Code”, starting on page 485. This chapter discusses Smart Code and contains a simple tutorial which shows you how to set up a Group in your design, generate “Get/Set” Smart Code and access the Group members independently of the toolkit you are using.
2. Chapter 17, “Thin Client Smart Code”, starting on page 501 describes how to create a thin client and a separate server application from your design. The server is a *CGI script* and the communication between the client and server is achieved by using the standards of the Internet. Again, a tutorial is included to familiarize you with the basic concepts.
3. Chapter 18, “Internet Smart Code”, starting on page 533 explains how to generate code form your design which is capable of accessing pages on the World Wide Web. A simple tutorial is included to let you try this out.

Creating a Group

To create a Group, select any number of widgets in your design and press the "Add to Group" toolbar button (shown in Figure 15-1) or select "Add to a New Group" from the Widget menu. The Group Editor appears, as shown in Figure 15-2.

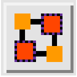
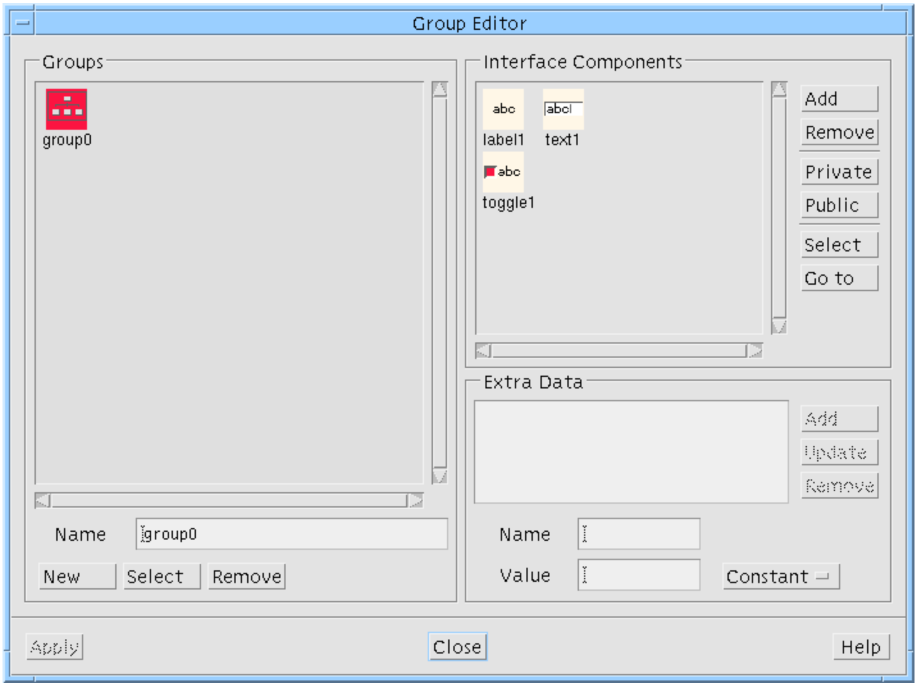


FIGURE 15-1 The "Add to Group" Toolbar Button



Name: group0

FIGURE 15-2 Group Editor

The Group Editor allows you to:

1. Change the name of the group.
2. Specify the Group members.

To change the name of the Group, select it, type the new name into the text field labelled "Name" beneath and press the Return key. This takes effect immediately.

When a Group is selected, its members are displayed in the list on the right. There are a number of functions which apply to the Group members:

1. Add. Add any widgets currently selected in the design area to this group.
2. Remove. Remove the selected members from the group.
3. Private. Define the selected member(s) as “private” to this group. This refers to the way code is generated for the group and is only relevant to thin client callbacks. More details are provided in “Public/Private Members” on page 480.
4. Public. Define the selected member(s) as publicly accessible (from a server). This refers to the way code is generated for the group and is only relevant to thin client callbacks. More information is provided in “Public/Private Members” on page 480.
5. Select. Select the corresponding widgets in the design area.
6. Go to. For this function, you must have only one member selected in the Group Editor. Pressing “Go to” makes the selected widget visible in the design area, unfolding nodes in the hierarchy if necessary.

Groups as Shortcuts

Groups come into their own when used for Smart Code. This is explained in Chapter 16, “Get/Set Smart Code”, starting on page 485. Within Sun WorkShop Visual, however, using groups provides an extra level of convenience. Three areas where groups can be used are detailed in the following sub-sections.

Fast Multiple Selection

In the Group Editor, the “Select” button underneath the list of Groups highlights all the widgets of the Group in the design area. This allows you to set up Groups of widgets which may need to be selected again and again. Using Groups in this way generates very little extra code. The Group is simply defined as an array of widgets.

Quick Find

The “Go to” button next to the list of Group members in the Group Editor, causes Sun WorkShop Visual to display the selected widget in the design area, unfolding nodes in the hierarchy if necessary. By making particular widgets into Groups by themselves, you can mark them for finding later.

Links

Groups can be used as a link destination in the Edit Links dialog, as shown in Figure 15-3.

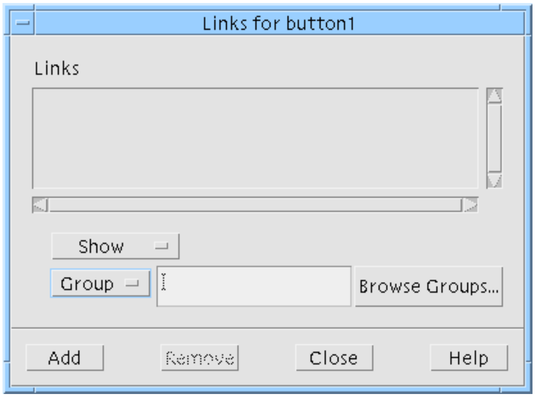


FIGURE 15-3 Edit Links Dialog

Using Groups as link destinations provides you with a quick and simple means of, for example, hiding or disabling whole groups of widgets at once.

Groups for Smart Code

Public and private members and Extra Data are features of the Group structure intended for development of thin client applications, which are explained in Chapter 17, “Thin Client Smart Code”, starting on page 501. You do not need to use these features if you are not using thin client or Internet Smart Code.

Public/Private Members

The “Public” and “Private” toggles allow you to control the accessibility of each member when code is generated. You only need to change this if you are setting up a Group to be used in a thin client/server application.

By default, members of a Group are defined to be public. This means that, when a Group is passed outside of the client application (to a remote server, for example) the receiving routine can access those members.

You may wish only the thin client of your application (which is the user interface) to have access to certain members of a group. The member should then be made "private". It is visible everywhere in the thin client application, but not in the server. Because the client application is controlling the user interface, it may well need access to more members than the server. This is discussed in more detail in "Extra Data - Function" on page 483.

Creating thin client and server applications is described in Chapter 17, "Thin Client Smart Code", starting on page 501.

Extra Data

The Group Editor contains an area labelled "Extra Data" where you may add extra members to a Group. These are intended for client/server transactions and represent name/value pairs. They are always treated as Strings.

Since only the Group is passed to a Smart Code callback, allowing extra data inside the Group provides a means of passing more information into the callback. This is especially useful when the callback is functioning in a separate server application and does not have direct access to the rest of the client application.

The extra data section of the Group Editor contains a list showing the existing definitions, two text fields for the name and value of the data and an option menu allowing you to choose whether the extra data is a "Constant", "Variable" or "Function".

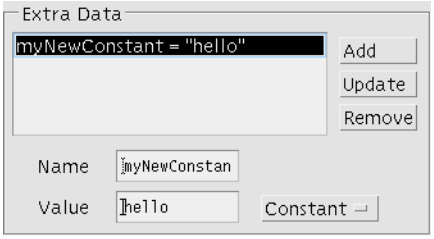
To add extra data, type in a name, a value, choose its type and then press "Add".

The extra data is treated in exactly the same way as any other member of a Group; there are get and set functions provided for them. This means something slightly different for a constant, a variable or a function. These "types", therefore, are described individually in the sub-sections below.

Extra Data - Constant

The extra data defined as shown in Figure 15-4 results in the extra member "myNewConstant" being initialized to the string "hello".

Routines to get and set the value "hello" are provided in the generated code. Chapter 16, "Get/Set Smart Code", starting on page 485 provides details on getting and setting the values of Group members.



Extra Data

myNewConstant = "hello"

Name: myNewConstan

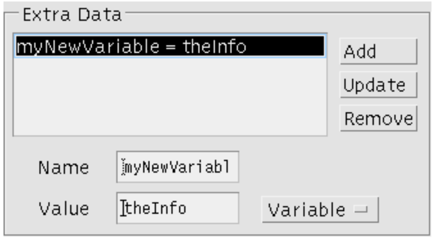
Value: hello

Constant

FIGURE 15-4 Extra Data - Constant Type

Extra Data - Variable

To add data with a variable value to your Group, use an existing variable as the “Value” and enter your own name into the “Name” field. An “extern” definition will be added to the generated code file for the existing variable (the one named in the “Value” field). An example of this is shown in Figure 15-5.



Extra Data

myNewVariable = theInfo

Name: myNewVariabl

Value: theInfo

Variable

FIGURE 15-5 Extra Data - Variable Type

You would have to ensure that “theInfo” is a string variable and is defined elsewhere. The get and set functions provided in this case would get and set the value of the variable. For example, supposing that you have defined “theInfo” like this:

```
char * theInfo = “hello”;
```

And you have defined “myNewVariable” as shown above. The following line would assign the string “hello” to `str`:

In C:

```
char * str = SC_GET(Value, myGroup->myNewVariable);
```

In C++:

```
char * str = myGroup->myMyNewVariable->getValue();
```

In Java:

```
String str = myGroup.myNewVariable.getValue();
```

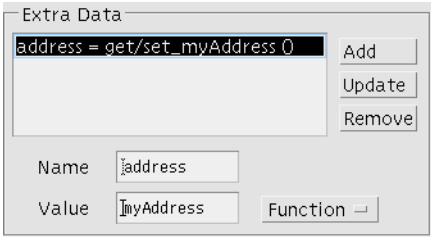
Chapter 16, “Get/Set Smart Code”, starting on page 485 provides details on getting and setting the values of Group members.

Extra Data - Function

The ability to add extra data in the form of functions provides added flexibility. Defining an extra member of type “Function” adds a member to the Group whose value is accessed via a getter and setter function filled in by you. The string typed into the “Name” field is the name of the extra member. The string typed into the “Value” field is used for the names of the get and set routines. For example, if you type `address` into the “Name” field and `myAddress` into the “Value” field, an extra member called “address” is added to the Group and its value is accessed via the routines:

- get_myAddress
- set_myAddress

This is shown in Figure 15-6.



address = get/set_myAddress ()

Name: address

Value: myAddress

Function

FIGURE 15-6 Extra Data - Function Type

Sun WorkShop Visual generates stubs for these two routines into the file in the callouts sub-directory whose filename is a concatenation of the name of the group and the “Value” field. For example, assuming that the “`myAddress`” routines shown above are defined in a group named “myGroup”, the file containing them would be named `myGroup_myAddress.c` (for C code).

The following line of C code fetches the value of “address”:

```
SC_GET(Value, group->address);
```

This line of code causes the new routine `get_myAddress` to be called. You should fill in `get_myAddress` so that a value is returned. As you might expect, “`SC_SET`” calls `set_myAddress`.

Note – You may type the same name into both the “Name” and the “Value” field. You may find this less confusing as the getter and setter routines will match the Group member name more closely.

Chapter 16, “Get/Set Smart Code”, starting on page 485 provides details on getting and setting the values of Group members.

Defining extra data in the form of functions is intended for assembling the data of a Group into a form expected by a server. For example, you may have a server which has no awareness of user interface components and which expects an address in the form of one long string. Your user interface, however, may contain several text fields for the user to type in an address. You could have an extra data member named “address” which is a function. The `get_address` routine (assuming the string “address” was typed into both the “Name” and “Value” fields) would return a concatenation of all the address fields. A callback sitting in the server could then simply fetch the value of “address” in the normal way:

```
char * value = SC_GET(myGroup->address);
```

Similarly, the corresponding `set_address` routine could take a string from the server and separate out the strings relevant to each text field. In this example, you may also wish to make the text field widgets “Private” and the “address” member “Public”, since the server is only interested in the one “address” member.

There are many circumstances in which the data available from the Group needs to be refined for the server. For example, you may wish to send only the selected text from a text area (as opposed to the whole text).

CHAPTER 16

Get/Set Smart Code

Introduction

Sun WorkShop Visual can generate toolkit independent code which it calls Smart Code. Smart Code can be used in a single generated application or as a means of communicating between a client and a server. When used for client server applications, Sun WorkShop Visual generates a *thin client* using Java, Motif (C or C++) or Microsoft Windows MFC code and a separate server application. Sun WorkShop Visual generates code which uses HTTP as the transport protocol. The generated application can be used within your organizations Intranet or it can connect to any location on the Internet.

To create a client server or “Web-aware” application from your design, Sun WorkShop Visual uses Groups as the basic data structure for communication and Smart Code as the means to access the Group. Groups also provide extra usefulness within your design. They are described in Chapter 15, “Groups”, starting on page 477.

How the Smart Code Information Is Organized

- This chapter describes the “Get/Set” flavor of Smart Code.
- “Get/Set Tutorial” on page 493 takes you through the steps for defining Groups and using Get/Set Smart Code.

- Chapter 17, “Thin Client Smart Code”, starting on page 501 describes how to tell Sun WorkShop Visual to split your design into a thin client and a server application.
- Chapter 18, “Internet Smart Code”, starting on page 533 describes how to build an application which is capable of fetching data from anywhere on the World Wide Web.

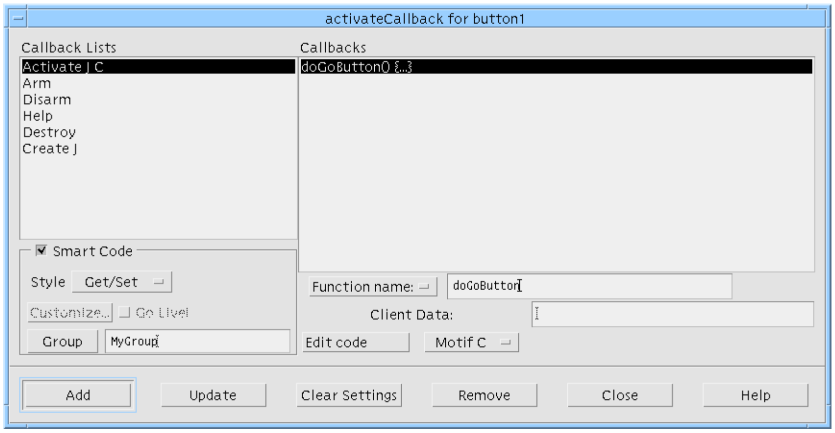
Using Smart Code

By default, traditional Motif, MFC or Java callbacks are generated. Ask for Smart Code by selecting the “Smart Code” toggle in the Callbacks dialog.

Smart Code gives your callback toolkit-independence, not language independence. You still choose the language you wish to generate. Smart Code is simply a layer of code which “wraps up” the widgets in a Group *using the specified language.*

An option menu, shown in Figure 16-1, allows you to choose one of three styles of Smart Code:

1. **Get/Set**. This provides you with a suite of “getter” and “setter” functions for a group of widgets.
2. **Thin Client**. This provides not only the getters and setters, it also triggers Sun WorkShop Visual to generate a separate client application containing the user interface and a server application containing the Smart Code callback(s). In addition, the code to handle the communication protocol between the two is generated.
3. **Internet**. This option is very similar to the “Thin Client” option above except that no server application is generated. Use this option for generating applications which will connect to an existing server or which will fetch Web pages.



☑ Smart Code

Style Get/Set

Group MyGroup

Function name: doGoButton

FIGURE 16-1 Smart Code in the Callbacks Dialog

A Smart Code callback requires a Group. This Group gives the callback a *handle* onto a set of objects which can then be programmed in a toolkit-independent way. To specify a Group do one of the following:

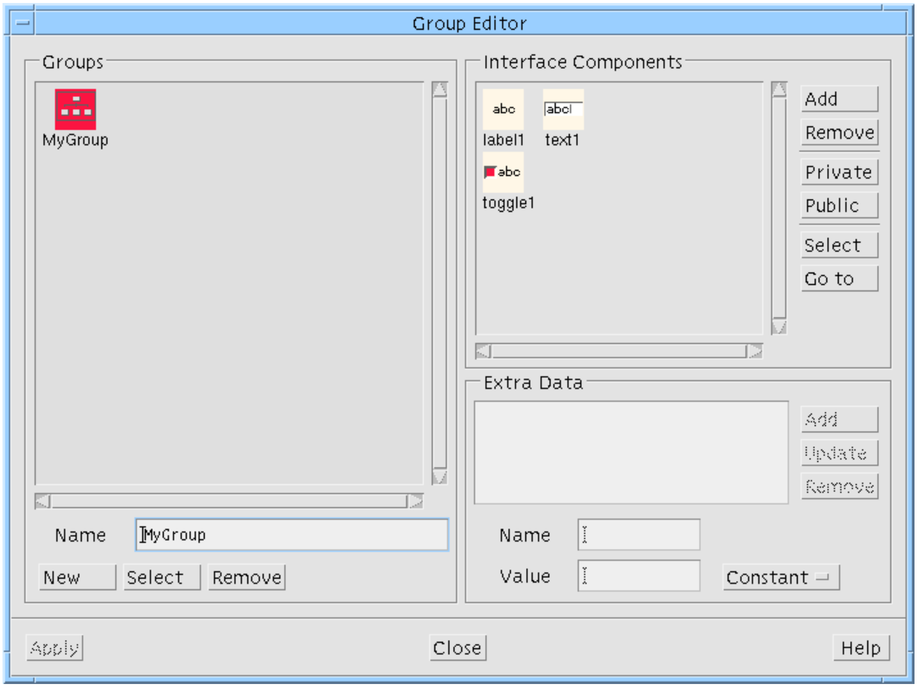
1. Type the name of a Group into the “Group” text field.
2. Press the “Group” button to display the Group Editor, select a Group and press “Apply”.

The “Style” option menu lets you choose the type of Smart Code you require and, therefore, the structure of the application you are creating. The code generated for each style is different. The sub-sections below describe each style individually.

Get / Set Smart Code

Choosing “Get / Set” from the Smart Code style menu in the Callbacks dialog tells Sun WorkShop Visual to generate *toolkit independent wrappers* for the components in the specified Group. These are also called “getter” and “setter” functions; they allow you to get and set the value of the Group’s components without writing any toolkit-specific code.

For example, suppose you are using C and you have a widget named “text1” and a toggle named “toggle1” in a Group named “MyGroup”, as shown in Figure 16-2.



Groups: MyGroup

Interface Components: label1, text1, toggle1

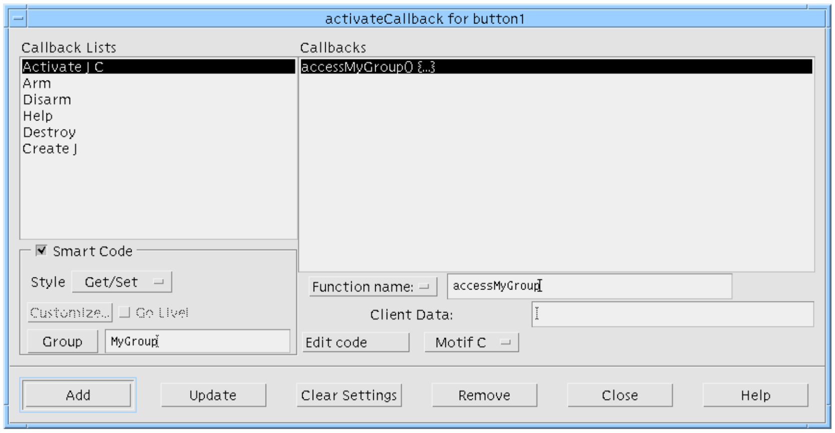
Name: MyGroup

Name:

Value:

FIGURE 16-2 Group “MyGroup” in Group Editor

In order to access this Group in a callback, you would define a Smart Code callback, choosing “Get/Set” as the Smart Code style and specifying “MyGroup” as the Group to be used in the callback. This is shown in Figure 16-3.



Callbacks: accessMyGroup() {...}

☑ Smart Code

Style: Get/Set

Group: MyGroup

Function name: accessMyGroup

Client Data:

Motif C

FIGURE 16-3 Callback Using “MyGroup”

Smart Code Callback in C

The code generated for the callback shown in Figure 16-3 would look like this:

```
#include <stdio.h>
#include “sc_groups_c.h”

void
accessMyGroup_user ( d, group, client_data)
        sc_data_t * d;
        MyGroup_t* group;
        void* client_data;
{
}
```

The data structure sc_data_t which is passed into Smart Code callbacks contains information mainly relevant to the communication between a client and a server. You can find the definition of this structure in the header file sc_groups_<language>.h. This file is generated by Sun WorkShop Visual into the directory specified in the Generate dialog. See “Generated Code” on page 525 for more information on the code generated for Smart Code and Chapter 17, “Thin Client Smart Code”, starting on page 501 for more information on creating a client and server from your design.

To get the value of “text1”, you would simply need to add the following line to this callback:

```
char * val = SC_GET(Value, group->text1);
```

This next piece of code checks the state of the toggle and enters text into “text1” if the toggle is set:

```
if (SC_GET(State, group->toggle1)
     SC_SET(Value, group->text1, “The toggle is set”);
```

This example assumes that you are using C. Because the C syntax is more cumbersome, macros are also provided to maintain ease of programming. The macros are called SC_SET and SC_GET. They take two parameters. The first is the type of query (the thing you are getting - Value, State etc.) and the second is the widget you wish to access (via its Group).

Smart Code Callback in C++

The Smart Code callback generated for the example shown in Figure 16-3 is different from the example above when C++ is the chosen language in the Generate dialog. In this case, Sun WorkShop Visual generates a subclass of the internal pre-defined Smart Code class sc_data_c using the name of the callback as the new class name. The class sc_data_c corresponds to the C data structure sc_data_t described above. The routine called from the main application when this callback is triggered, is the doit() method of the new class. The following listing of accessMyGroup_user.cpp from the directory callouts_cpp illustrates this. accessMyGroup was the name of the callback method entered in the Callbacks dialog.

The getGroup() method is a convenience routine in the parent sc_data_c class:

```
#include <stdio.h>
#include “sc_groups_cpp.h”

class accessMyGroup_user: public sc_data_c
{
        void doit() {
           MyGroup_c * g = (MyGroup_c *)getGroup();
Add your own code here  →
        }
};

sc_data_c *
getNew_accessMyGroup()
{
        return (sc_data_c*) new accessMyGroup_user;
}
```

In C++, you do not need to use the macros described for C code. The following line of code fetches the contents of “text1”:

```
char * str = g->text1->getValue();
```

The following line checks the state of the toggle and sets the contents of “text1” if the toggle is set:

```
if (g->toggle1->getState())
    g->text1->setValue(“The toggle is set”);
```

Smart Code Callback in Java

When Java is the selected language in the Generate dialog, your Smart Code callback shown in Figure 16-3 is generated as a subclass of the SCData Smart Code class. This is similar to the C++ model described above. You add your own code to the `doit()` method of the class. The super class, SCData, includes a convenience function to get a handle to the Group. The SCData class is defined in the file `SCData.java` in the `utils_java` subdirectory beneath your generate directory.

For the example shown in Figure 16-3, the following code would appear in a file named `accessMyGroup_user.java` found in the `callouts_java` subdirectory:

```
package callouts_java;


import groups_java.* ;
import utils_java.* ;


public class accessMyGroup_user extends SCData
{
        public void doit() {
                group0_c g = (group0_c)getGroup();
Add your own code here  →


    }

        public static accessMyGroup_user getNew()
        {
                return  new accessMyGroup_user();
        }
}
```

In Java, you do not need to use the macros described for C code. Instead, you would do something similar to the C++ model. The following line of code would fetch the contents of “text1”:

```
String str = g.text1.getValue();
```

The following line checks the state of the toggle and sets the contents of “text1” if the toggle is set:

```
if (g.toggle1.getState())
    g.text1.setValue(“The toggle is set”);
```

Priming Dialogs Using the Create Callback

The example above shows a callback defined for “Activate”. This means that your “Get/Set” code is not called until the button is pressed. If you need to “prime” the widgets in a dialog when the dialog is first displayed, define a Smart Code callback for the “Create” callback of the shell. This callback will occur just once when the dialog is created..

The create callback of a widget is called after all the widget’s children have been created. The widgets in a Group must be created before a Smart Code callback can access them. Using the shell’s create callback to prime a dialog means that you can be sure all the widgets have been created before the create callback is triggered since the shell is the last widget to be created. If you do not wish to set up a callback on the shell, add an arbitrary container to your design (if you do not already have one) which *contains* all the widgets in your Group. Since child widgets are always created before their parent, in this way you can be sure that the widgets in the Group you have specified for the Smart Code callback have been created before the Smart Code callback is triggered.

Using Getters and Setters

The toolkit-independent wrappers are very simple to use. A full list of the functions provided for each widget type is given in Appendix C, “Getters and Setters”, starting on page 855.

HTML Files

In order to help you locate callbacks and familiarize yourself with the files generated when Smart Code is requested, Sun WorkShop Visual also generates a set of HTML files. The main one is:

- index.html

and is found in the top-level directory (i.e. the directory named in the Generate dialog).

This file, when opened in a web browser (or anything else which can read HTML) lists the files generated along with a brief description of each one. Sun WorkShop Visual creates a symbolic link in your generate directory to a directory in the Sun WorkShop Visual install directory in order to access HTML files which describe the unchanging features of Smart Code.

Get/Set Tutorial

This simple tutorial takes you through the main features of Groups and Get/Set Smart Code. The tutorial sets up a Group and uses Smart Code to get and set the values of the group members.

When completed, the application looks as shown in Figure 16-4.

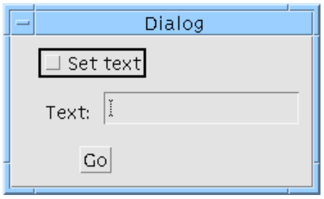


FIGURE 16-4 Final Application

1. **Start Sun WorkShop Visual.**
2. **Create an Application Shell and a Form. Inside the Form, put a button, a label, a textfield and a toggle.**

   Figure 16-5 shows this initial hierarchy.

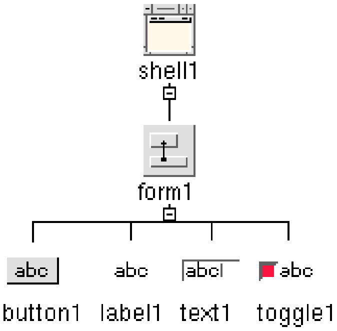


FIGURE 16-5 Initial Tutorial Hierarchy

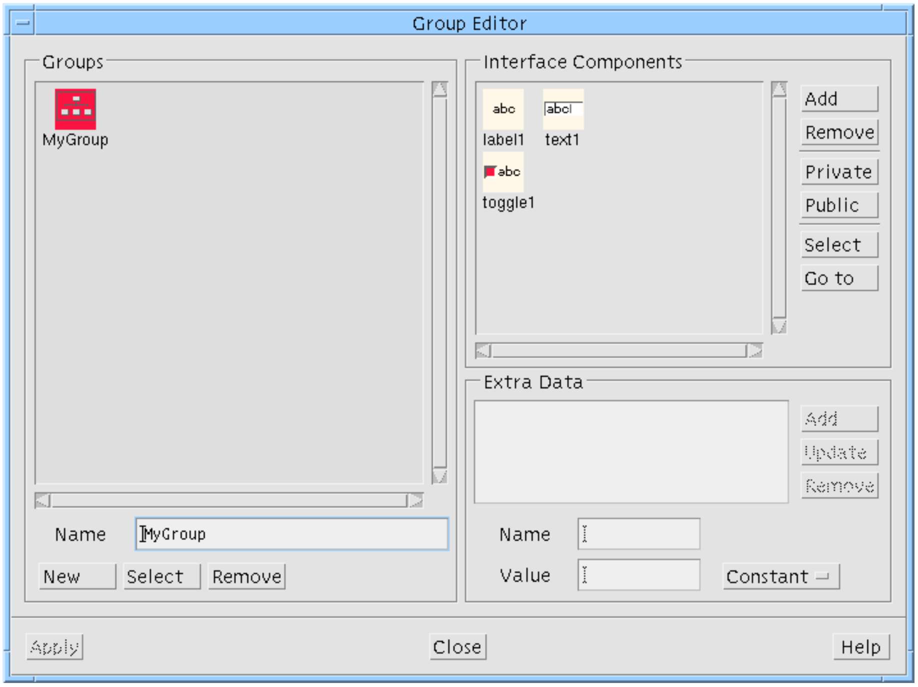
3. **Select the label, text and toggle widgets and press the Group Editor button on the toolbar.**

   This is the same as selecting “Add to a New Group” from the Widget menu.

   The Group Editor now appears, as shown in Figure 16-6.

4. **Change the name of your new group to “MyGroup” by typing this into the text field labelled “Name” and pressing the Return key.**

   You do not have to change the name of the group, but doing so may be simpler, especially if you intend to have a large number of groups.



Name: MyGroup

FIGURE 16-6 Group Editor for New Group

5. **Close the Group Editor.**

   The “Apply” button only becomes enabled when you are selecting a group to add to a callback, as will be seen later.

6. **Change the layout of the widgets in the Form and the labels of the button, label and toggle so that they are as shown in Figure 16-7.**

   This is a purely cosmetic alteration. The label of the button is “Go”, the label is set to “Text:” and the toggle is “Set text”.

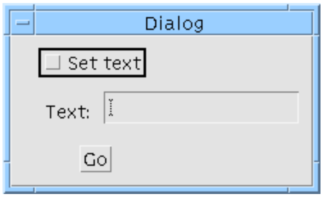


FIGURE 16-7 Widgets Laid Out and Labels Set

7. **Display the Callbacks dialog for the button.**

   You can do this by either pressing the Callbacks button on the toolbar or choosing “Callbacks” from the Widget menu.

8. **Make sure that “Activate” is selected from the callback lists.**

   We are choosing “Activate” so that the callback is called when the button is pressed.

Note – In the Callback Lists section, the Activate callback is shown with a ‘J’ after it indicating that it is applicable to Java code. These annotations are described in “Callbacks” on page 172.

---

9. **Set the “Smart Code” toggle.**

10. **Select “Get/Set” from the Smart Code option menu.**

11. **Press the button labelled “Group”.**

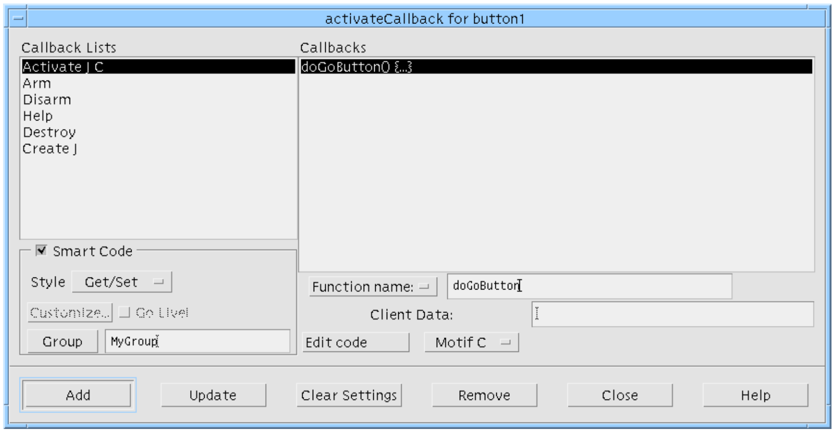
    This displays the Group Editor. Note that the “Apply” button is now enabled.

12. **Select MyGroup and press “Apply”.**

    The Editor is dismissed and the name “MyGroup” is added to the text field next to the Group button. You could also have typed the name in directly.

13. **Name the callback “doGoButton” and press “Add”.**

    The new callback is shown in the list with curly braces enclosing periods ({...}) to indicate that this is a Smart Code callback. Figure 16-8 shows the defined callback.



Activate | C
Arm
Disarm
Help
Destroy
Create |

doGoButton() {...}

☑ Smart Code

Style: Get/Set

Function name: doGoButton

Client Data:

Group: MyGroup

Motif C

FIGURE 16-8 New Smart Code Callback

14. Close the Callbacks dialog.

The definition of the callback is complete. Now we need to fill it in.

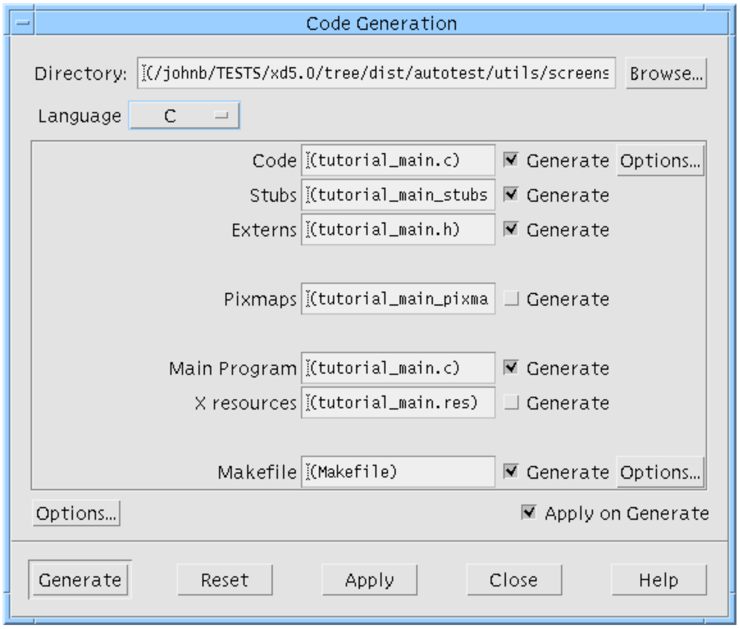
15. Save your design as `tutorial.xd`**.**

16. Display the Generate dialog and choose a target directory for your source files.

The target directory is shown at the top of the dialog.

17. Make sure that the "Stubs", "Code", "Externs", "Main program" and "Makefile" generate toggles are set.

The correct configuration of the Generate dialog is shown in Figure 16-9. For this tutorial we have changed the names of the files from the default ("untitled") to "tutorial".



Directory: /johnb/TESTS/xd5.0/tree/dist/autotest/utils/screens

Language C

Code tutorial_main.c ☑ Generate

Stubs tutorial_main_stubs ☑ Generate

Externs tutorial_main.h ☑ Generate

Pixmaps tutorial_main_pixma □ Generate

Main Program tutorial_main.c ☑ Generate

X resources tutorial_main.res □ Generate

Makefile Makefile ☑ Generate

☑ Apply on Generate

FIGURE 16-9 Generate Dialog

18. Press the "Options..." button above "Generate" to display the Code Generation Options dialog.

19. Set the "Strings" option menu to "Code" and close this dialog,

For a simple tutorial, it is easier to generate label strings into the code file than to generate a separate resource file.

20. Now generate C code by pressing the "Generate" button.

Of course, you can generate C++ code. The example listings below use C.

The directories and files shown in Figure 16-10 are generated.

Tip – "Get/Set Generated Code" on page 526 provides more information on the files generated for "Get/Set" Smart Code.

---

21. Save your design.

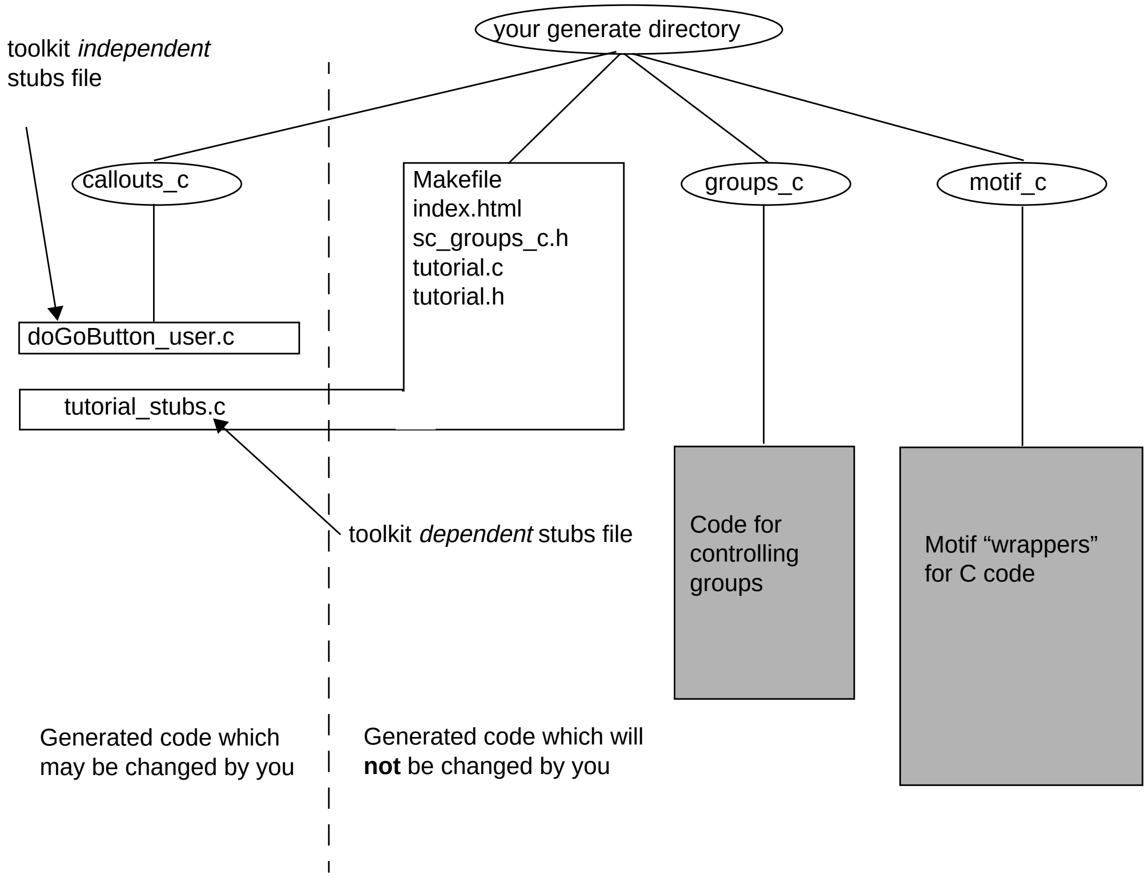

your generate directory
toolkit *independent* stubs file
callouts_c
Makefile
index.html
sc_groups_c.h
tutorial.c
tutorial.h
groups_c
motif_c
doGoButton_user.c
tutorial_stubs.c
toolkit *dependent* stubs file
Code for controlling groups
Motif “wrappers” for C code
Generated code which may be changed by you
Generated code which will **not** be changed by you

FIGURE 16-10 Files and Directories Generated For Tutorial

22. **Use your favorite editor to edit the file** `doGoButton_user.c` **in the directory** `callouts_c.`

    This is the file highlighted in Figure 16-10.

23. **The callback itself is also called “doGoButton_user”.**

    This is listed below:

```
#include <stdio.h>
#include “sc_groups_c.h”

void
doGoButton_user ( d, group, client_data)
        sc_data_t * d;
        MyGroup_t* group;
        void* client_data;
{

}
```

Note – This routine is called from the button activate callback in `tutorial_stubs.c`.

24. **Add the following lines to “doGoButton_user”:**

```
if (SC_GET(State, group->toggle1))
    SC_SET(Value, group->text1, “The toggle is set”);
else
    SC_SET(Value, group->text1, “”);
```

These few lines simply check whether the toggle is set or not. If it is set, then the text field displays “The toggle is set”. If it is not, the text field is cleared.

25. **Save your edits.**

26. **Make sure that you have the environment variable VISUROOT set to Sun WorkShop Visual’s install directory and that you have your C compiler in your PATH.**

See the Generating Code chapter in the main User Guide for more information on setting up prior to compiling.

27. **Type:**

```
    make
```

in the directory where the files were generated.

28. **Once your application has built, try running it.**

29. **Press the “Go” button with and without the toggle set.**

The tutorial is now complete. Although the functions you have added are very simple, you have established a framework within which your application can be extended very easily. It would now be much simpler to generate both a client and a server from your design by defining a Thin Client Smart Code callback. This is described in Chapter 17, “Thin Client Smart Code”, starting on page 501. A step-by-step tutorial shows you how to do this in “Thin Client Smart Code Tutorial” on page 504.

CHAPTER **17**

Thin Client Smart Code

Introduction

Motif and MFC applications are traditionally large scale applications. Modern Internet technologies encourage the *thin client* approach. This is an application structure with a clean division between:

- A lightweight client controlling just the user interface.
- A remote server providing data back to the client.

Note – The server application generated by default by Sun WorkShop Visual is a CGI program using a Web server to communicate with the client.

This chapter describes how Sun WorkShop Visual helps you to migrate, very simply, from a large scale application to the structure described above. This is achieved by first “grouping together” widgets to create portable data structures, then “enhancing” your callbacks so that they can either run remotely or communicate with a remote server. There are no other changes you need to make to your design.

Sun WorkShop Visual also lets you try out the connection between the client and the server dynamically *before you generate code*. Setting the “Go Live!” toggle for a thin client or Internet callback activates that callback by connecting to the specified URL. This is described in “Going Live” on page 521.

You will need to understand how to use both Groups and Get/Set Smart Code in order to use the thin client (or Internet) Smart Code because Groups, along with their getters and setters, are the nuts and bolts of all types of Smart Code. Information on these subjects is found in:

1. The grouping together of widgets is described in Chapter 15, “Groups”, starting on page 477.

2. Chapter 16, “Get/Set Smart Code”, starting on page 485 describes the Get/Set Smart Code which provides you with toolkit-independent wrappers for the widgets in your design.

Using Thin Client Smart Code

Creating a thin client and server application from your design is done by setting up a special Smart Code callback. To do this, you only need to do the following:

1. Display the Callbacks dialog for a widget in your design.
2. Select the “Smart Code” toggle.
3. Choose “Thin Client” from the Smart Code option menu.
4. Specify a Group.
5. Name and add the callback.

What a Group is, and how to create one, is described in Chapter 15, “Groups”, starting on page 477.

Smart Code gives your callback toolkit-independence, not language independence. You still choose the language you wish to generate. Smart Code is simply a layer of code which “wraps up” the widgets in a Group *using the specified language.*

There are three types of Smart Code:

1. **Get/Set**. This provides you with a suite of “getter” and “setter” functions for a group of widgets. This is described in Chapter 16, “Get/Set Smart Code”, starting on page 485.
2. **Thin Client**. This provides not only the getters and setters, it also triggers Sun WorkShop Visual to generate a separate thin client application containing the user interface and a server application containing the callback you are defining. In addition, the code to handle the communication between the two is generated. This chapter describes the use of thin client Smart Code.
3. **Internet**. This also provides the getters and setters and tells Sun WorkShop Visual to generate a client application which is ready to communicate with a remote server. The code for communicating with the server is also generated. Internet Smart Code is described in Chapter 18, “Internet Smart Code”, starting on page 533.

Specifying a Group

Both “Thin Client” and “Internet” Smart Code styles require the name of a Group to be passed to the callback. The Group gives the callback access to data in the user interface, even though the callback may be running in a remote server. Either type the name of the Group directly into the “Group” text field or press the “Group” button. This displays the Group Editor. Select the Group you require and press “Apply” to both dismiss the Editor and place the name of the Group into the text field.

Getters and Setters

All of the stubs generated by Sun WorkShop Visual when either “Thin Client” or “Internet” is selected, have full use of the “getter” and “setter” routines for the specified Group. The Get/Set Smart Code is described in Chapter 16, “Get/Set Smart Code”, starting on page 485. The callback in a server has a slightly different access to the Group components. It calls a single get or set routine which accesses a default “value”. This “value” differs according to the type of widget. For example, a text field has a string value (its contents) and a toggle has a boolean value (its state).

Customize Button

Sun WorkShop Visual Selecting “Thin Client” or “Internet” from the Smart Code “Style” option menu in the Callbacks dialog enables the “Customize” button. This button displays the Customize dialog, allowing you to configure how data is sent to and received from the server. Sun WorkShop Visual does provide sensible defaults. You will always have to change the URL address of the server with which you are communicating, however.

The Customize dialog appears slightly different according to whether it is displayed with “Thin Client” or “Internet” selected from the option menu. The only difference is the label for the Send custom data handler text
box. This is because Sun WorkShop Visual uses a different HTTP method for the two types of application. When “Thin Client” is selected, Sun WorkShop Visual uses the POST protocol. By default, Sun WorkShop Visual uses the GET protocol when “Internet” is selected because it is assumed that you are only fetching web pages. You can change this to use POST by providing a custom Send data handler. More about this is given in “Custom Data Handlers” on page 518.

The Customize dialog is explained in “Customizing the Server Connection” on page 514 below.

See “Thin Client/Internet Generated Code” on page 527 for details of the code that is generated when a server callback is defined, including a list of the stubs files.

Go Live

The “Go Live!” toggle becomes sensitive when a thin client or internet callback has been defined. Setting the toggle enables you to test the connection between the client and the server from within Sun WorkShop Visual, before generating code. Detailed information on this is given in “Going Live” on page 521.

Requirements

Both thin client and Internet Smart Code styles use the standards of the World Wide Web as the means of communication between the generated client application and a server. In order to run the generated and compiled applications and to use the “Go Live” feature, you will need:

1. A networked computer connected to an Intranet or Internet.

2. A running Web server.

If you need more information on this, consult your systems administrator or contact your Sun WorkShop Visual supplier.

Thin Client Smart Code Tutorial

This simple tutorial gives you the chance to try out the “Thin Client” style of Smart Code. By following these steps you will quickly generate a thin client containing the user interface shown in Figure 17-1 and a server which sets the contents of the text field according to whether or not the toggle is set. The server can run remotely on your own intranet or anywhere on the Web.

The initial steps are covered only briefly as it is assumed that you are already familiar with the general use of Sun WorkShop Visual. If you are not, you may wish to try the main Sun WorkShop Visual tutorial which starts in Chapter 2 ”Building the Widget Hierarchy“.

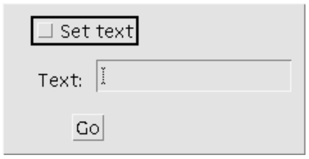


FIGURE 17-1 Thin Client Application User Interface

1. **Start Sun WorkShop Visual.**
2. **Create a simple hierarchy: application shell->form->button, label, toggle, text field.**

   This is shown in Figure 17-2.

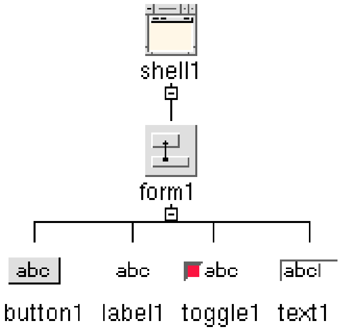


FIGURE 17-2 Hierarchy for Smart Code Tutorial

3. **Label the button “Go”, the toggle “Check me” and the label “Text:”.**
4. **Give the shell the title “Tutorial Example”.**

   Do this in the shell’s resources dialog.
5. **Display the Form Layout Editor and lay out the widgets so that they look roughly as shown in Figure 17-3.**

   This is a purely cosmetic change. It affects only the appearance and has no impact on the function of the application.

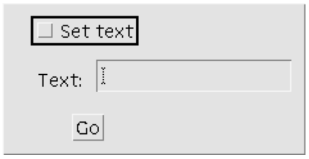


FIGURE 17-3 Tutorial Widgets Laid Out

6. **Make the toggle and the text field a Group called “MyGroup”.**

   If you can’t remember how to do this, see “Creating a Group” on page 478.

Tip – There is a lot more information about Groups in Chapter 15, “Groups”, starting on page 477. The “Get/Set Tutorial” on page 493 is a simple example which shows you how to create and use a Group.

---

7. **Select the button and display the Callbacks dialog.**

8. **Type “doGoButton” into the function name text box.**

   This callback will, in effect, *move* to the server-side of your application.

9. **Set the Smart Code toggle button.**

10. **Change the Smart Code style to “Thin Client”.**

    When you do this, the “Customize” button is enabled.

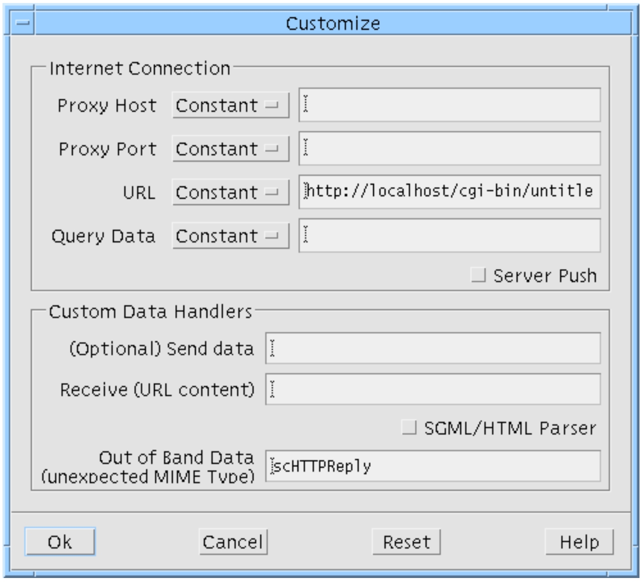
Tip – Much more detail concerning the use and meaning of “Thin Client” is given in "*Thin Client Server Callbacks*" section on page 512.

---

11. **Press the “Customize” button to display the Customize dialog.**

    The dialog is shown in Figure 17-4.

Tip – The various fields in the Customize dialog are described in "*Customizing the Server Connection*" section on page 514.



Customize

Internet Connection

Proxy Host Constant

Proxy Port Constant

URL Constant http://localhost/cgi-bin/untitle

Query Data Constant

☐ Server Push

Custom Data Handlers

(Optional) Send data

Receive (URL content)

☐ SGML/HTML Parser

Out of Band Data (unexpected MIME Type) scHTTPReply

FIGURE 17-4 Customize Dialog

12. **If you are behind a firewall, fill out the proxy and port fields.**

    If you are not sure about this, check with your system administrator or see how your web browser is configured.

Tip – "*More on Proxies*" section on page 516 lists ways of finding out what your proxy is.

---

13. **Make sure that the URL field refers to the name and location of your server program.**

    For example:

    ```
    http://localhost/cgi-bin/untitled
    ```

    would refer to a server named “untitled” located in the “cgi-bin” directory of your intranet. Following this tutorial, your server application will be called “tutorial” so you should use this name here - simply change “untitled” to “tutorial”. “cgi-bin” is a standard directory name, “localhost”, however, should be changed to the name of your intranet.

Note – In order to work out where to put your server on your filesystem, check the way your Web server has been configured with respect to the physical location of the “cgi-bin” area of the server.

---

14. **Press “Ok” in the Customize dialog.**

    For this tutorial, we are going to use the default data handlers so we do not need to change them. Pressing “Ok” closes the dialog.

15. **Back in the Callbacks dialog, press the “Group” button and select “MyGroup” from the Group Editor.**

    You can type the name of the group directly into the text box, but selecting it from the Group Editor avoids typing mistakes.

16. **Press “Add” to add your “doGoButton” function to the list of callbacks.**

17. **Close the Callbacks dialog.**

18. **Save your design as “`tutorial.xd`”.**

    It is good practice to save your design periodically.

19. **Display the Generate dialog by selecting “Generate” from the Generate menu.**

20. **Choose “C” from the Language menu.**

21. **Choose a target directory for your files.**

22. **Make sure that the “Stubs”, “Code”, “Externs”, “Main program” and “Makefile” generate toggles are set.**

    The “Stubs” file is not selected by default, so make sure it is set before generating.

23. **Press the “Options” button at the bottom of the Generate dialog to display the Code Options dialog.**

    Do not confuse this with the Primary Source File Options Dialog which is displayed when pressing the “Options” button next to the “Code” file.

24. **In the Code Options dialog, change the “Strings” option menu to “Code”.**

    By default, this is set to “Resources”. For this simple tutorial, it is easier to generate the strings (which is all the labels you have set) into the code. This affects only the appearance of the final dialog.

25. **Press “Ok” in the Code Options dialog to save your change and dismiss the dialog.**

26. **Back in the Generate dialog, press “Generate”.**

    As well as generating the main code files, Makefile and HTML index in the top-level directory, Sun WorkShop Visual also generates the following sub-directories:

    - **callouts_c**. This contains the client-side stubs for Smart Code callbacks.
    - **server_c**. This contains all the code for your server including the Smart Code callback stub.
    - **groups_c**. This contains the code for the Groups.

- **motif_c**. This contains the “wrappers” which make Group components independent of Motif.
- **http_c**. This contains all the code for communication between the client and the server using the HTTP protocol.

Figure 17-5 shows all the files generated. The files containing stubs are highlighted.

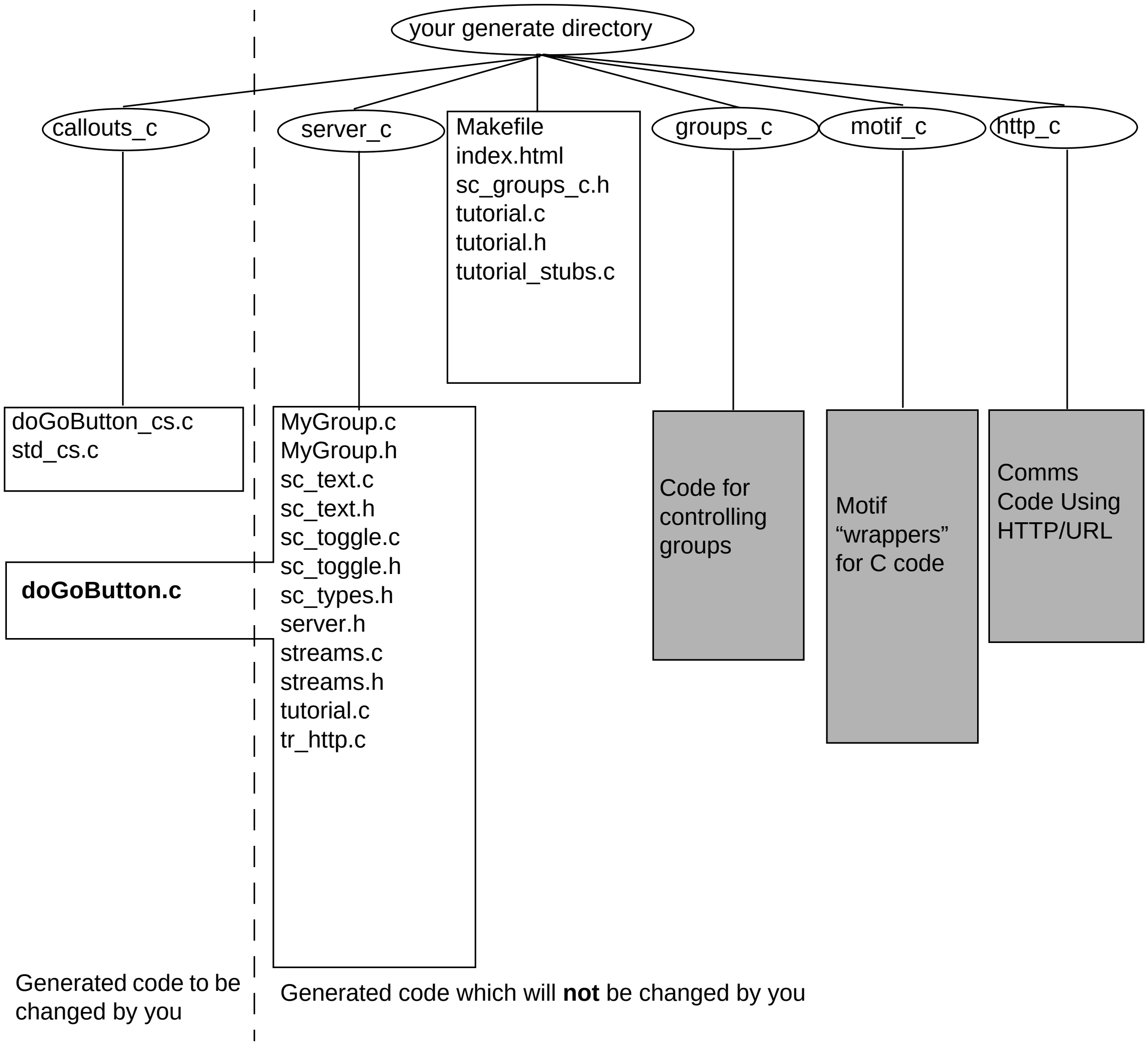


FIGURE 17-5 Files and Directories Generated For Tutorial

27. **Save your design.**

The rest of this stage of the tutorial takes place outside of Sun WorkShop Visual. You can exit Sun WorkShop Visual if you wish.

Of the generated files, there are three you need to know more about. These are:

- `callouts_c/doGoButton_cs.c`. This file is in the client side of your application. It contains the “precondition” and “postcondition” routines which are called directly before and after the server callback is called. By default they are empty. You only need to fill them in if, for example, you wish to preprocess or postprocess the group structure being sent to the server.
- `callouts_c/std_cs.c`. This file contains the standard error handler for handling out of band data. It also contains the routine which communicates with the server. It is this routine which calls the precondition and postcondition routines. Although this file is always the same, it is placed here so that, if necessary, you can see for yourself how the handshaking is performed. You would not normally have to change this file.
- `server_c/doGoButton.c`. This file contains the Smart Code callback stub which will be called from the client. It is residing in the server application.

Tip – "*Generated Code*" section on page 525 contains more information on the code generated by Sun WorkShop Visual for thin client and Internet Smart Code.

---

28. **Edit the file** `doGoButton.c` **in the “server_c” directory.**

This contains the Smart Code callback stub. You need to add code here to make the server perform its function.

29. **Change the method “doGoButton_callback” so that it contains the lines indicated below:**

```
int
doGoButton ( sc_data_t * data)
{
          MyGroup_t * g = (MyGroup_t*)data->group;

```

Add these lines

```
        if (SC_GET(g->toggle1))
                SC_SET(g->text1, “Server says: toggle is set”);
        else
                SC_SET(g->text1, “Server says: toggle is NOT set”);
```

```

        return 1;
}
```

30. **Make sure you have the environment variable VISUROOT set to Sun WorkShop Visual's install directory and your C compiler is in your PATH.**

    See the Generating Code chapter in the main User Guide for more information on setting up prior to compiling.

31. **At the command prompt, type**

    ```
    make
    ```

    to build the client application and:

    ```
    make server
    ```

    to build the server application.

    You can take the entire "server_c" directory somewhere else to compile. Take a copy of the Makefile too. The Makefile assumes that the directory "server_c" is beneath it.

32. **Move the server application (named "`tutorial`" in the "server_c" directory) to the location named in the URL field, as in Sun WorkShop Visual Replay.**

33. **Try running the client application (named "`tutorial`" in the generate directory).**

    Press the button both with and without the toggle set. A message is displayed in the text box. This message is produced by the server, which is checking the value of the toggle first. Although this is a simple example, it demonstrates a structure which can easily be expanded.

Going a Step Further

Having completed this tutorial, you may like to try some more advanced features of thin client and Internet Smart Code. Provided as part of your Sun WorkShop Visual package are HTML files containing instructions for running supplied Sun WorkShop Visual Replay scripts which run the tutorials for you. You simply watch it running and then examine the results. Open the following files in an HTML browser:

1. `$VISUROOT/lib/locale/<YourLocale>/sc/timex.html`. This describes the "Server Push" tutorial, demonstrating how to create an application with automatic remote update.

2. `$VISUROOT/lib/locale/<YourLocale>/sc/parsex.html`. This describes how to create an application which fetches a Web page and then parses it.

Note – VISUROOT is the install directory of your Sun WorkShop Visual and "YourLocale" is the name of the locale you are using.

Thin Client Server Callbacks

When you generate code from a design containing a “Thin Client” Smart Code callback, Sun WorkShop Visual divides the generated code into a thin client application containing all the user interface code and a server containing the Smart Code callback. This server is a *CGI script* and therefore uses the standards of the World Wide Web as its means of communication. The code for communicating between the client and the server, using HTTP, is generated too.

Figure 17-6 gives a diagram of the structure of the application Sun WorkShop Visual generates when “Thin Client” callbacks are defined. In addition to the Smart Code callback in the server, stubs are generated on the client side of the application to allow you to do any necessary operations immediately before and after data is sent to the server.

The Smart Code callback residing in the server has a handle to a Group. This is the Group specified when the callback was defined. The Group provides a means of passing data between the client and server and gives the server some access to the interface components.

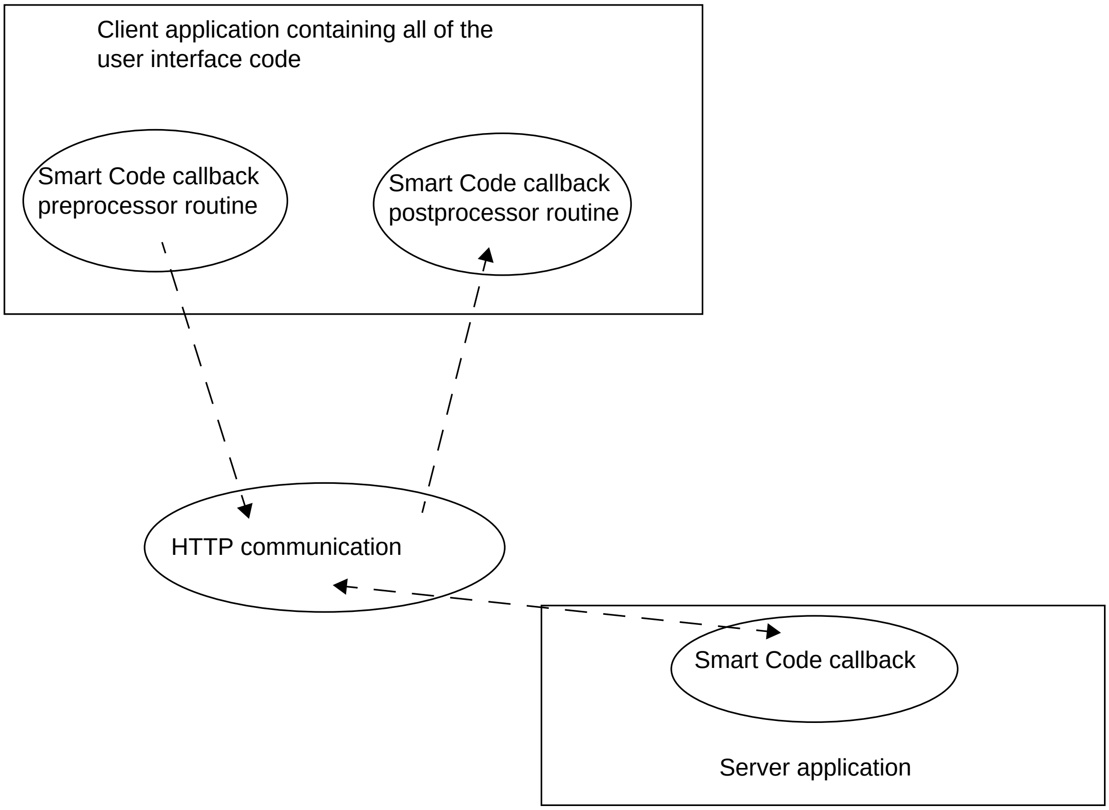

Client application containing all of the user interface code
Smart Code callback preprocessor routine
Smart Code callback postprocessor routine
HTTP communication
Smart Code callback
Server application

FIGURE 17-6 Thin Client Callback Application Structure

Server Application

The server application generated by Sun WorkShop Visual when thin client Smart Code is requested, is a *CGI script*. This means that it is a program, which can be written in any language (or shell script), which is accessed by means of the HTTP standard. The URL of your server application is passed from the client application to your Web server which then launches your server application. Output from the server application is returned to the client. The server application is generated in the language you request in the Generate dialog.

If you do not have a running Web server, your client and server applications will not be able to communicate. The Internet Smart Code also assumes that you have a Web server up and running in order to access Web pages on the Internet. Contact your

Sun WorkShop supplier for information about Sun’s Web server products or visit Sun’s Web page at http://www.sun.com. There are a number of freely available Web servers available on the Internet; you should be able to locate one with a quick Web search.

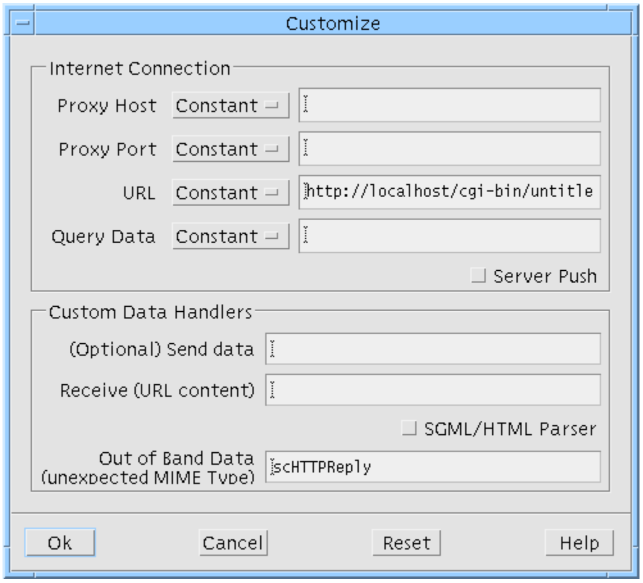
Customizing the Server Connection

Pressing the “Customize” button for “Thin Client” or “Internet” Smart Code styles in the Callbacks dialog displays the Customize dialog shown in Figure 17-7.

This dialog allows you to specify how data is sent to and received from the server. There are two main areas in this dialog:

1. Connections. The details of the network connections, such as the proxy and the URL.
2. Custom data handlers. The names of routines which should be used for communicating across the network instead of those supplied by Sun WorkShop Visual.

These are described in separate sections below.



Customize

Internet Connection

Proxy Host: Constant

Proxy Port: Constant

URL: Constant — http://localhost/cgi-bin/untitle

Query Data: Constant

☐ Server Push

Custom Data Handlers

(Optional) Send data:

Receive (URL content):

☐ SGML/HTML Parser

Out of Band Data (unexpected MIME Type): scHTTPReply

FIGURE 17-7 Customize Dialog

Connections

Sun WorkShop Visual uses the information in this section to generate code to establish a link across a network. The fields in this dialog are:

1. **Proxy Host**. This is only required if you are behind a firewall. It is the name of the conduit between your computer and the internet.

2. **Proxy Port**. This is only required if you are behind a firewall. It is the port number to use on the proxy host.

Note – If you are not sure what values to set for proxies, see "More on Proxies" on page 516.

---

3. **URL**. This is the only field in the dialog which *must* be filled in. It is a string which describes the location of an HTML document or CGI script. What this "document" is depends on the *style* of communication you have chosen (as described above):
   - If you have chosen thin client Smart Code, the URL probably names a CGI program which is the server side of your application.
   - If you have chosen Internet Smart Code but have not specified a Receive Handler, the URL is the name of a completely separate remote server from which you only expect some acknowledgment of receipt.
   - If you have chosen Internet Smart Code and you have specified a Receive Handler, the URL probably names a completely separate remote server (or file on that server) from which you are expecting data in a particular format.

URLs are part of the established World Wide Web protocol.

4. **Query Data**. This is a *query string*. When a question mark (?) appears in a URL, the remainder is assumed to be an algorithm for matching a document or its contents, such as may be expected by a search engine. It is up to the server to interpret this string. Sun WorkShop Visual appends the question mark character and then the query string to the URL.

5. **Server Push**. This is a toggle which, when selected, tells Sun WorkShop Visual that you are prepared to accept *asynchronous* input from the server. This means that data will be coming into the client application when it is ready rather than after a request.

URL Library

The URL library used by, and supplied with, Sun WorkShop Visual has been modeled on the Java URL and URLConnection classes in order to facilitate the transition to Java and to maintain a common interface whether you are using C, C++ or Java. This library is available for your use, should you wish to maintain full control over the connection between client and server. The programming interface for the URL library is described in the files reached from symbolic links inside the generated `index.html` file. This file is found in your generate directory, as described in "HTML Files" on page 530.

If you have not yet generated code, open the following file in a HTML browser:

```
$VISUROOT/lib/locale/<your_locale>/sc/URL.html
```

where VISUROOT is the install directory of your Sun WorkShop Visual and `<your_locale>` is the name of the locale you are using. If you are unsure about your locale, try typing `locale` into a terminal window. This prints out your locale information. Use the string assigned to LANG. Example locales are:

- C (for English).
- ja (for Japanese).

More on Proxies

A network proxy is a conduit linking your computer (or your Intranet) to the Internet through a firewall. A firewall is a "security screen" which prevents unauthorized access to your computer from the outside world. In order to connect to the internet from behind a firewall, you will have to specify a proxy host and a proxy port. If you don't already know what these are, you could try the following options:

1. Ask your systems administrator. Someone must have configured your network, email and internet connection.

2. Find out how your Web browser is configured because, in order to connect to the Internet, it also has to use a proxy. More about this in the following paragraph.

Somewhere in your Web browser, under "Options" or "Preferences" or "Configuration", there will be a dialog where you can specify your network preferences. This dialog allows you to tell the browser whether or not it needs to use a proxy and, if so, what it is. There are a number of options here:

1. No proxy is used. In this case, you do not need to provide one in Sun WorkShop Visual's Customize dialog.

2. A proxy is specified manually either in this dialog or a sub-dialog. If this is the case, use the proxy specified. If there is a choice of proxies, use the HTTP proxy.

3. The URL of a configuration file is specified for automatic setting of the proxy. In this case, check the proxy specified in the configuration file.

Constant, Variable, or Function

“Proxy Host”, “Proxy Port”, “URL” and “Query Data” each have a corresponding option menu allowing you to define whether the value you enter is a “Constant”, “Variable” or “Function”. As its name suggests, a constant simply retains the value typed. Selecting “Variable” causes Sun WorkShop Visual to declare an external variable with the specified name. You should define the variable in your own code.

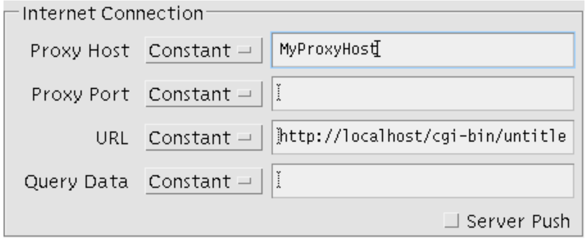
Selecting “Function” causes Sun WorkShop Visual to generate a file in the “callouts” sub-directory using the name of the function as the filename. This file contains two functions: a “get” and a “set”. These routines contain a commented spaceholder where you should insert your own code for getting and setting the proxy host.

For example, Figure 17-8 shows a proxy host defined as a function called “`MyProxyHost`”. Assuming C code, Sun WorkShop Visual would generate the following two routines in the file named `MyProxyHost.c`:

```
static char * MyProxyHost_value = (char*)0;
char *
get_MyProxyHost ( AnyGroup_t* ingroup)
{
    group0_t * group = (group0_t*)ingroup;
    if (!MyProxyHost_value) {
          /* do something */
          (void) fprintf( stderr, “Warning: getMyProxyHost()
                                   returns NULL\n”);
    }
    return MyProxyHost_value;
}

void
set_MyProxyHost (AnyGroup_t* ingroup, char* value)
{
    group0_t * group = (group0_t*)ingroup;
    /* set MyProxyHost_value here */
}
```

The comments indicate where you should add code to get and set the proxy host. You may, for example, have defined a form in your user interface which allows a user to enter a proxy host into a text field. You might then get the contents of the text field and assign it to the “`MyProxyHost_value`” shown above.



Proxy Host: Constant — MyProxyHost

Proxy Port: Constant —

URL: Constant — http://localhost/cgi-bin/untitle

Query Data: Constant —

□ Server Push

FIGURE 17-8 “MyProxyHost” as a Function

The bibliography lists books which you may find useful in connection with networking and protocols.

Custom Data Handlers

The section of the Customize dialog labelled “Custom Data Handlers” gives you the option of overriding the following default routines:

1. Send. This is the routine which sends data to the server

2. Receive. This is the routine which receives data from the server.

3. Out of Band Data. This is the routine which handles data of an unexpected type returning from the server.

Leaving an empty text box for the Send or Receive handlers causes Sun WorkShop Visual to generate a default routine. For Out of Band Data, the name of the default routine appears in the text box. These defaults provide basic functionality, allowing you to create a working client-server application without immediately having to provide your own handlers. The following sub-sections describe briefly what the default routines give you as well as information to help you write your own.

Send Handler

By default, Sun WorkShop Visual generates a routine which sends data to the server. This routine does the following:

- Opens a print stream to the server.
- Prints the MIME header.
- Prints each component of the group as a serialized HTTP object.
- Prints the MIME end.

If you have specified a send handler in the Customize dialog, the routine described above will not be called. Instead, your routine is called. Your routine is defined as returning an integer. It takes two parameters, in the following order:

- A pointer to the output stream.
- A pointer to the group.

All of the communication protocol remains in place when you override the send routine - it is only the sending of data which is replaced. The default send routine is still generated, but not called, when you override it. If you wish to look at it to see how it sends the MIME header, MIME end and how it serializes the group components, it is called `httpSendOutputStream` and is found in:

```
<YourGenerateDirectory>/http_c/tr_http.c
```

Your send handler is placed in the callouts directory. See "Generated Code" on page 525 for more details on the code generated and where it is located.

Tip – If you generate code from a design using the Get/Set shorthand notation described in "Go Live" on page 504, an example Send Handler is generated. You may wish to use this as a starting point.

Receive Handler

If you specify a receive data handler in the Customize dialog, this routine is called in addition to all of the communication protocol which Sun WorkShop Visual generates by default.

In C and C++ code, your routine is defined as returning an integer and taking two parameters:

1. A pointer to a structure defined by Sun WorkShop Visual called `sc_stdcs_t` for C code and `sc_stdcs_c` for C++. This structure gives your routine access to the Group. There are also private fields which are for internal use only. It is defined in the file `sc_groups_c.h` in the directory where you are generating code.

2. A pointer to a structure defined by Sun WorkShop Visual called sc_idata_t in C and sc_idata_c in C++. This structure gives your routine access to the input stream, its length and MIME type. It is defined in the file `http_transport.h` in the `http` sub-directory.

For Java, the Receive Handler you specify is made a subclass of the Sun WorkShop Visual defined class, SCInputDataHandler. Add your own code to the doit() method. This method is "void" and takes one parameter which is an instance of the SCIData class, as defined in `SCIData.java` in the `utils_java` sub-directory.

The receive handler supplied by default ignores the returned data. You will almost certainly wish to parse the returned data. Sun WorkShop Visual is supplied with a full HTML parser. This parser is simple to use because it is data driven - that is, in the same way that widgets register an interest in the callbacks they wish to see, you can tell the parser which HTML tags you are interested in. This is all explained in detail in "Extracting Information from HTML Data" on page 541.

Your receive handler is placed in the callouts directory. See "Generated Code" on page 525 for more details on the code generated and where it is located.

Tip – If you generate code from a design using the Get/Set shorthand notation described in "Go Live" on page 504, an example Receive Handler is generated. You may wish to use this as a starting point.

Out of Band Data Handler

Sun WorkShop Visual expects incoming data to conform to the MIME standard. If unexpected data is received, a routine in the client side of the application, generated by Sun WorkShop Visual, is called. This default routine, called `scHTTPReply`, simply prints the incoming data onto standard output. It is generated for you (into the callouts subdirectory) if you have a thin client or Internet Smart Code callback defined. It is generated in the language requested in the Generate dialog. This is the routine in C code:

```
int
scHTTPReply ( csdata, data)
        sc_stdcs_t* csdata;
        sc_idata* data;
{
     extern DataInputStream * newDataInputStream();

     InputStream* i = (InputStream*)(*data->getInputStream)( data);
     DataInputStream *  d = newDataInputStream( i);
```

```
    char * s;

    printf(“data from server:\n”);
    if (!d)  {
        printf(“no data\n”);
        return 0;
    }
    while ((s = (*d->readLine)( d)) != (char*)0) {
        printf(“%s\n”, s);
    }
    (*d->delete)( d);

    printf(“end data\n”);
}
```

If you specify your own routine as the “Out of Band Data” handler in the Customize dialog, the routine is defined in the same way as the default. The parameters it takes are exactly the same as those listed for the Receive Handler, which is described above.

For Java code, the format of the “Out of Band Data” handler is exactly the same as the Receive Handler - it is a subclass of the Sun WorkShop Visual defined class, SCInputDataHandler. Add your own code to the doit() method.

Your “Out of Band Data” handler is placed in the callouts directory. See “Generated Code” on page 525 for more details on the code generated and where it is located.

Going Live

The “Go Live!” toggle lets you turn the Sun WorkShop Visual dynamic display into a fully functional thin client interface. You can set the toggle for both thin client and Internet callbacks.

You must define and add the callback before the “Go Live!” toggle becomes enabled. For thin client Smart Code, the callback is the name of a routine in the server. For Internet Smart Code, it is the name of a callback in your client application.

Sun WorkShop Visual provides a shorthand getter or setter to allow you to create dynamic customization in the Customize dialog. This shorthand uses the convention of spreadsheet packages (the ‘@’ sign) to indicate “the contents of...”. For example, the following:

```
@text1
```

means “the contents of the widget named text1 in the specified Group”. The specified Group is the one named in the callbacks dialog. This notation fetches the contents of a control when it is used in any of the “Connections” fields (URL and Proxies). When it is used in the Receive handler field, it is assumed that you wish to *set* the contents of text1 to the data received from the server.

Go Live Tutorial

The following example illustrates “Go Live!”. It uses the World Wide Web to access a remote website and read a Web page. You will need to be running a Web server in order to complete this tutorial.

1. **Create the hierarchy shown in Figure 17-9. This is a shell->form->{button, textfield, text}.**

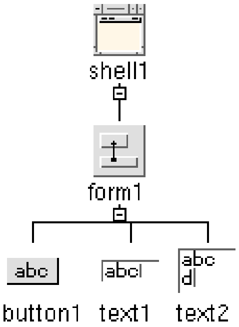
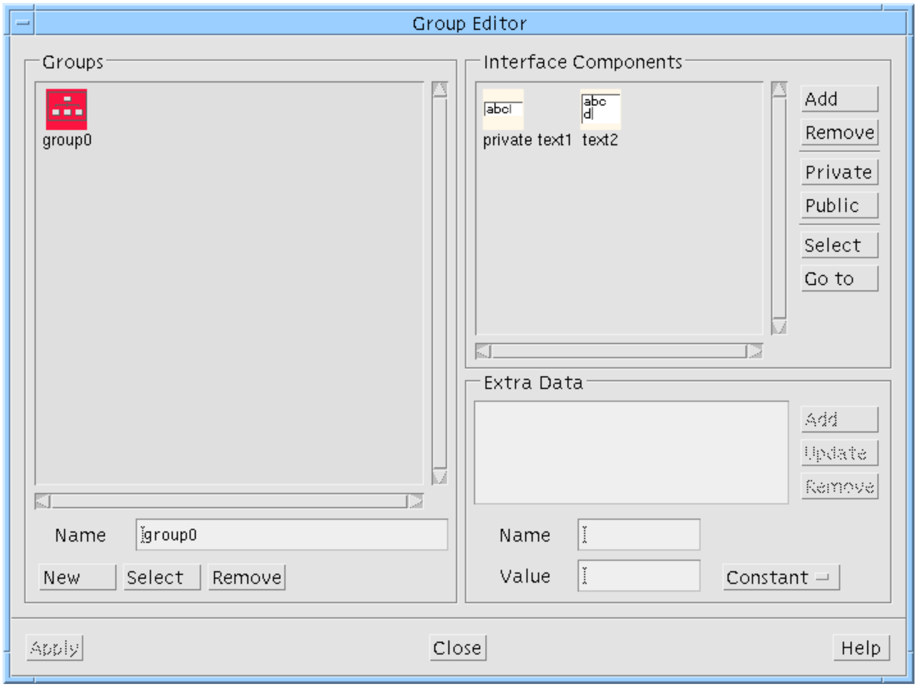


FIGURE 17-9 Hierarchy for Go Live Example

2. **Create a Group, called Group0, containing the textfield and text widget, making the textfield private.**

   This is shown in Figure 17-10.



Groups: group0

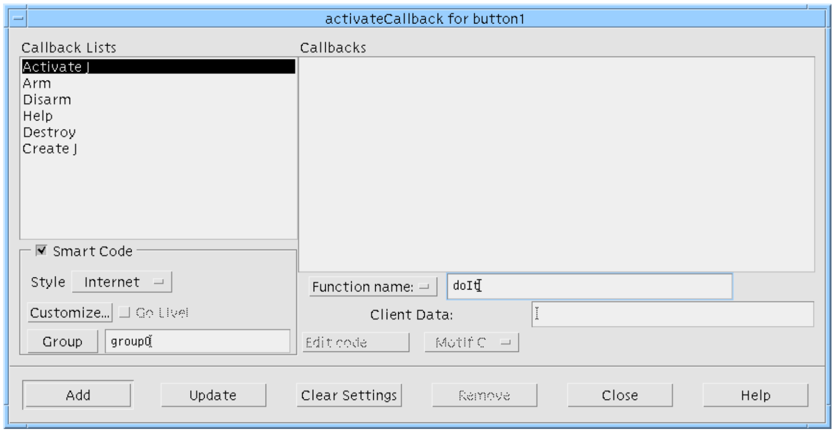
Interface Components: private text1, text2

Name: group0

FIGURE 17-10 The Group for the Go Live Example

3. **Display the Callbacks dialog for button1.**
4. **Set up an Internet Smart Code callback called "doit", as shown in Figure 17-11.**

   Don't add it yet as we need to customize it first.



☑ Smart Code

Style: Internet

Function name: doIt

Group: group0

FIGURE 17-11 The Server Callback for the Go Live Example

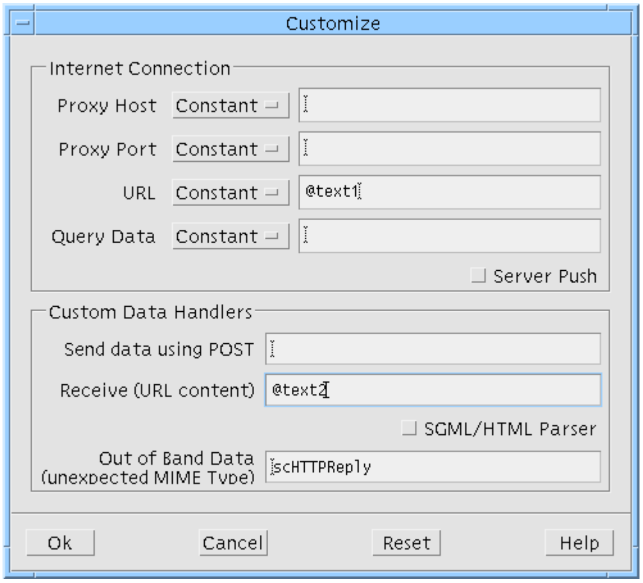
5. **Press the “Customize” button.**

6. **In the Customize dialog, put the getter shorthand notation for the contents of widget text1 into the “URL” field, as shown in Figure 17-12.**

   This shorthand notation is: “`@text1`”.

7. **Put the shorthand notation for the widget text2 into the “Receive Handler” field, as shown in Figure 17-12.**

   This shorthand is: “`@text2`”.



Proxy Host: Constant

Proxy Port: Constant

URL: Constant @text1

Query Data: Constant

Receive (URL content): @text2

Out of Band Data (unexpected MIME Type): scHTTPReply

FIGURE 17-12 The Customize Dialog for the Go Live Example

Tip – If you are behind a firewall, you will also have to set the Proxy fields. See “More on Proxies” on page 516 for details.

---

8. **Press “Ok” to save your settings and close the Customize dialog.**

9. **Back in the Callbacks dialog, set the “Go Live!” toggle.**

10. **Add the new callback and close the Callbacks dialog.**

11. **In the dynamic display’s textfield (the one named “text1”), type the following URL:**

    http://www.sun.com/workshop

12. **Press button1 in the dynamic display.**

    The main page of the Website typed into the textfield is displayed in text2.

13. **Try other URLs in text1 and press button1 again.**

    As long as you type a valid URL, text2 will display the HTML page to which the URL refers. For example, try typing the URL of your Intranet, if you have one.

    This example connects to a real server dynamically from within Sun WorkShop Visual.

Generated Code

Code files providing toolkit-independent wrappers and the definitions of Group objects are generated when Smart Code is selected from the Callbacks dialog. More files are generated for “Thin Client” and “Internet” than for “Get/Set” because, in addition to the getters and setters for the Group, all the code for the communication with a server is generated and, in the case of “Thin Client”, the server side of your application too.

Some of these files are in specially named directories which are created by Sun WorkShop Visual and which appear beneath the directory chosen in the Generate dialog. Other files appear in the chosen directory. Figure 17-5 on page 509 shows the files generated for the thin client Smart Code tutorial. This gives you an idea of the files and subdirectories that are generated.

Most importantly, all your callbacks or other routines which may need editing (such as Custom Data Handlers, pre- and post-processing routines, Extra Data functions etc.) are kept together so that you may find them easily. You do not need to change any other code files.

Exactly what is generated differs according to whether you have requested only “Get/Set” Smart Code for your callbacks or you have asked for “Thin Client” or “Internet”. The code generated in each case is described below.

Groups

If you are using groups but not Smart Code, an array of widgets is defined in the primary source file.

Without Smart Code, however, only the files described in the main User’s Guide “Generated Code” chapter are generated and no new directories are created.

Get/Set Generated Code

When generating code files, Sun WorkShop Visual uses an extension appropriate to the chosen language:

1. **`.c`** for C code
2. **`.cpp`** for C++ code
3. **`.java`** for Java code

The following examples use ".c" simply as an illustration.

If you have selected the "Get/Set" style of Smart Code for one or more callback, the following files are generated into the directory specified at the top of the Callbacks dialog:

- **Makefile**. This is a standard Sun WorkShop Visual-generated makefile which takes account of the new subdirectories.
- **sc_groups_c.h**. This header file simply includes the appropriate header files from the relevant subdirectory. It also defines the sc_data_t data structure, a pointer to which is passed into Smart Code callbacks and data handlers.
- **untitled.c**. This is the main source code file.
- **untitled.h**. This is the main header file.
- **untitled_stubs.c**. This is where your callback stubs are generated.
- **index.html**. This is an HTML file which lists and briefly explains all of the generated files. Use this file (by opening it in a web browser or other HTML reader) to find your way around the files and subdirectories.

In addition, the following sub-directories are created:

- **callouts_c**. This contains a file for each "Extra Data" defined for a group which is defined as a function.
- **groups_c**. This directory contains a source, header and HTML file for each group you have defined. It also contains a more general header file.
- **motif_c**. This contains the Motif "wrappers" for each component in your group(s). This directory also contains an HTML file for each group.

Note – The "c" at the end of each directory name shows that C was the language chosen for code generation. If another language is chosen this is replaced accordingly.

Where to Add Your Own Code for Get/Set

With “Get/Set” Smart Code, the following stubs files are generated:

- **<progname>_stubs.cpp** (**<progname>_stubs.c**). This is the traditional stubs file containing any callbacks you have defined. The stubs in this file have a strong connection with the toolkit and do not have access to any Groups or Smart Code data. Add only toolkit-specific code here. This file is generated into the directory specified in the Generate dialog.
- **<callbackname>_user.cpp (<callbackname>_user.c)**. This appears in the “callouts” sub-directory. When using Get/Set Smart Code, this contains the main callback stub, also named `<callbackname>_user`. This callback is free of any toolkit-specific code.
- **Extra Data function files**. This refers to the “Extra Data” area in the Group Editor. If you have added extra data and defined it to be “Function”, a file is generated using the name of the extra data as the filename. Each such file contains two routines: `get_<extradata>` and `set_<extradata>` (where `<extradata>` is the name supplied by you in the Group Editor). For C++ and Java these routines are methods of a class defined using the name of the “Extra Data” function. Information on using these routines is given in “Extra Data - Function” on page 483.

Figure 16-10 on page 498 shows a graphical representation of the files generated for the “Get/Set” Smart Code tutorial.

Thin Client/Internet Generated Code

When you generate code from a design containing any Smart Code callbacks, all of the files and directories described in “Get/Set Generated Code” on page 526 above are generated. In addition, the following directories are created by Sun WorkShop Visual:

- **http_c**. This contains the source code for parsing the URL and for sending and receiving data using the HTTP protocol.
- **server_c**. This is the server-side of your application.

The directory “server_c” contains all the files you need to build the server which will connect to the thin client application containing your user interface. The whole of this directory can be taken away to be built and run on a remote web server.

The Makefile in the top-level directory contains the rules for building both the client (your design) and the server. To build the client, you need only type:

```
make
```

at the command line. To build the server, type:

```
make server
```

at the command line.

Note – If you move the server code elsewhere, remember to take the Makefile too.

Where to Add Your Own Code for Thin Client/Internet

The following types of stub can be generated when you create a thin client or Internet Smart Code callback. Some of these are optional, depending on whether you have specified any routines in the Customize dialog and in the Group Editor:

- Toolkit-dependent client callback
- Toolkit-independent client callback
- Server callback (for thin client Smart Code only)
- Precondition and postcondition routines
- Internet connection functions
- Group Extra Data functions

These are described separately below.

Tip – See "Custom Data Handlers" on page 518 for details on adding your own data handlers and "Extra Data - Function" on page 483 for information on adding extra functions to the Group.

Toolkit-Dependent Client Callbacks

The file which contains your traditional, toolkit-dependent *client* callbacks is:

- **untitled_stubs.c** (or **untitled_stubs.cpp** for C++ code generation).

This is the callbacks file generated when *any* callback is defined, regardless of whether Smart Code is being used or not. When you define a thin client or Internet Smart Code callback, Sun WorkShop Visual automatically generates a client side callback in this file. This contains toolkit-dependent code. You should only add code here which uses the toolkit.

Toolkit-Independent Client Callbacks

The file which contains your toolkit-independent *client* callbacks is:

- **<callback>_cs.c** (or **<callback>_cs.cpp** for C++ code generation).

For each thin client or Internet Smart Code callback, a file is generated into the callouts subdirectory using the name of the callback as the filename with the addition of "`_cs`" (standing for client server). The file contains the callback routine (or method in the case of C++ and Java). This contains toolkit-independent code with access to the Group.

Server Callbacks

Your server callbacks are located in:

- server_c. There is a source file in this sub-directory for each callback you have defined. The file is given the same name as the callback.

Internet Connection Functions

A file is generated for each of the "Connections" in the Customize dialog, if "Function" is selected. Each file contains a `get_<connection>` and `set_<connection>` routine. The file is generated into the "callouts" subdirectory, using the name of the function that you specify as the filename.

A file is also generated for any custom data handlers specified in the Customize dialog. The filename is the name you type into the Customize dialog.

Extra Data Functions

As described in "Where to Add Your Own Code for Get/Set" on page 527, stub routines are generated for any "Extra Data" functions defined in the Group Editor.

Figure 17-5 on page 509 shows a graphical representation of the files generated for the "Thin Client" Smart Code tutorial.

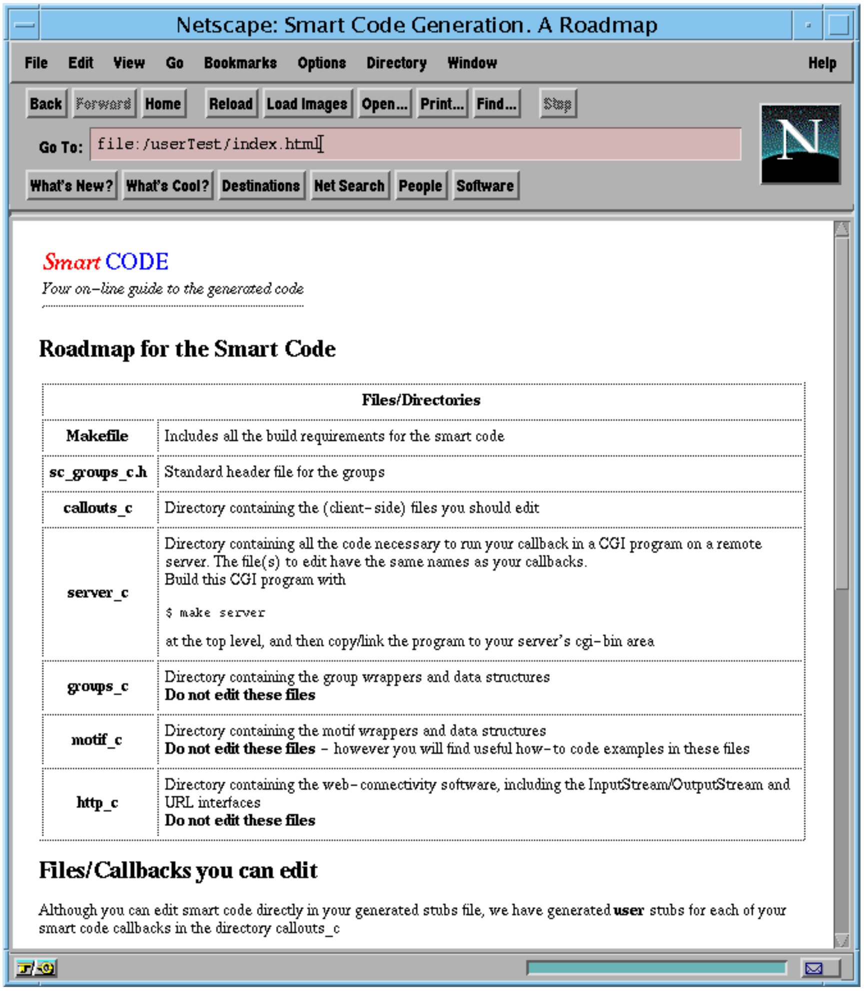
HTML Files

In order to help you locate callbacks and familiarize yourself with the files generated, Sun WorkShop Visual also generates a set of HTML files. The main one is:

- index.html

and is found in the top-level directory (i.e. the directory named in the Callbacks dialog).

This file, when opened in a web browser (or anything else which can read HTML) lists the files generated along with a brief description of each, as shown in Figure 17-13.



Smart CODE

Your on-line guide to the generated code

Roadmap for the Smart Code

Files/Directories	
Makefile	Includes all the build requirements for the smart code
sc_groups_c.h	Standard header file for the groups
callouts_c	Directory containing the (client-side) files you should edit
server_c	Directory containing all the code necessary to run your callback in a CGI program on a remote server. The file(s) to edit have the same names as your callbacks. Build this CGI program with `$ make server` at the top level, and then copy/link the program to your server's cgi-bin area
groups_c	Directory containing the group wrappers and data structures **Do not edit these files**
motif_c	Directory containing the motif wrappers and data structures **Do not edit these files** - however you will find useful how-to code examples in these files
http_c	Directory containing the web-connectivity software, including the InputStream/OutputStream and URL interfaces **Do not edit these files**

Files/Callbacks you can edit

Although you can edit smart code directly in your generated stubs file, we have generated **user** stubs for each of your smart code callbacks in the directory callouts_c

FIGURE 17-13 index.html in Netscape

There are hypertext links to other files, also generated by Sun WorkShop Visual, which describe the code generated for each group and each Motif “wrapper”.

This file also contains a link to the online reference material for Smart Code programming. You can view this reference material directly by opening the following file in an HTML browser:

```
$VISUROOT/lib/locale/<YourLocale>/sc/index.html
```

where VISUROOT is the install directory of your Sun WorkShop Visual and `YourLocale` is the locale you are using. If you are unsure about your locale, try typing `locale` into a terminal window. This prints out your locale information. Use the string assigned to LANG. Example locales are:

- C (for English).
- ja (for Japanese).

CHAPTER 18

Internet Smart Code

Introduction

Choosing “Internet” Smart Code for a callback causes Sun WorkShop Visual to generate a client application from your design. Internet Smart Code programming is about accessing pre-existing Web pages and CGI programs on public servers across the World Wide Web. The Smart Code callback appears in the client application (in a sub-directory named “callouts”). Figure 18-1 shows the structure of an application generated by Sun WorkShop Visual when an “Internet” callback is defined. Unlike thin client Smart Code, only the client application and communication code is generated for this type of callback.

You will need to understand how to use both Groups and Get/Set Smart Code in order to use Internet (or thin client) Smart Code because Groups, along with their getters and setters, are the nuts and bolts of all types of Smart Code. Information on these subjects is found in:

1. The grouping together of widgets is described in Chapter 15, “Groups”, starting on page 477.

2. Chapter 16, “Get/Set Smart Code”, starting on page 485 describes the Get/Set Smart Code which provides you with toolkit-independent wrappers for the widgets in your design.

The “Go Live” feature allows you to use Sun WorkShop Visual’s dynamic display as a prototype client in order to test your interface on live data as you are developing it. The tutorial starting on page 536 shows you how to do this and how to generate the application using a very simple example.

The use of “Go Live” for Internet Smart Code prototyping is limited because you will need to write your own Receive Handler to process and act on the incoming data.

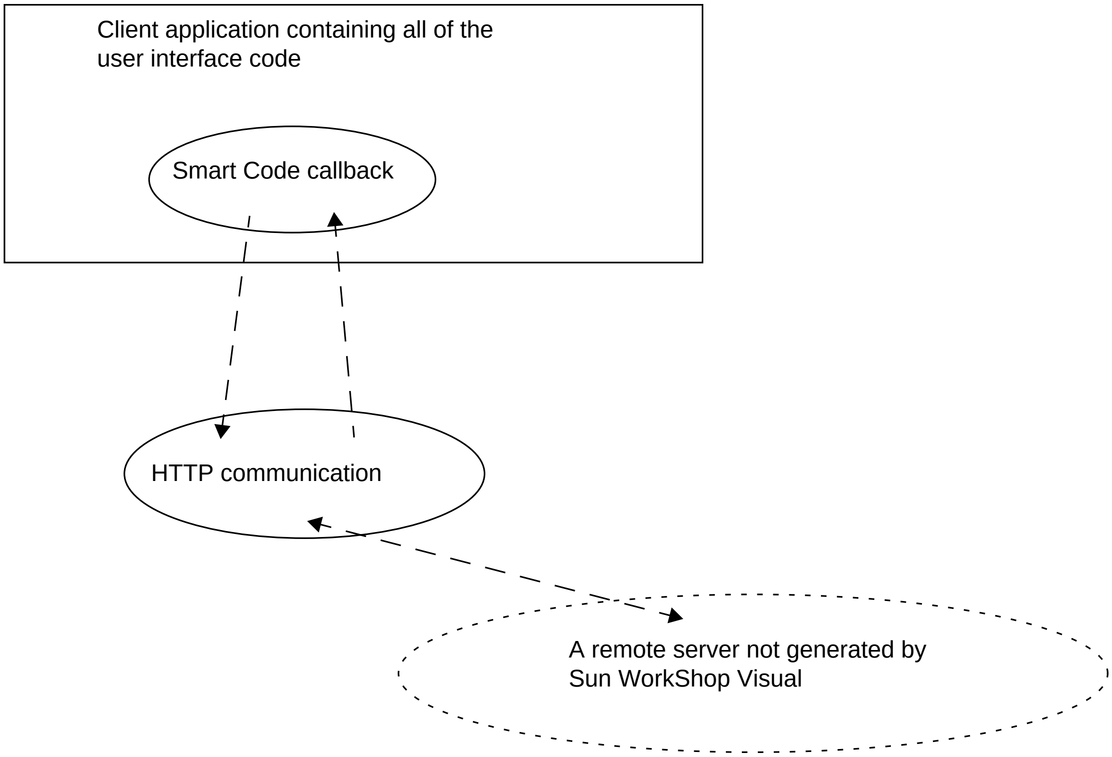


FIGURE 18-1 Internet Callback Application Structure

Internet and Thin Client

Internet and thin client Smart Code are similar:

- They both use the toolkit-independent getters and setters.
- They both use HTTP (generated through the Sun WorkShop Visual URL API) as the means of communication.
- They both make your design into a client application.

These similarities mean that parts of their user interface within Sun WorkShop Visual are shared. In order to use Internet, you may need to refer to the following sections which can be found in Chapter 17 ”Thin Client Smart Code“:

- “Customizing the Server Connection” on page 514.
- “Going Live” on page 521.
- “Generated Code” on page 525.

Internet is, however, distinct from thin client:

- For Internet Smart Code, no server application is generated.
- The GET HTTP protocol is used for Internet Smart Code (rather than POST).
- Because Internet designs are assumed to be *thick* clients, the way you structure them will be different.

Applications generated with Internet Smart Code might be used to:

- Fetch and parse the contents of a World Wide Web page.
- Connect to a pre-existing remote server
- Communicate with a server generated from another Sun WorkShop Visual design.

Receiving Data

For applications generated with Internet Smart Code you will need to provide a Receive Handler. Sun WorkShop Visual gives you a pointer to any data returned, but it is up to you to handle that data. Data handlers are part of the Customize dialog and described in “Customizing the Server Connection” on page 514. To make use of the data returned, Sun WorkShop Visual provides a library which allows you to express an interest in particular features of the input stream and then “pick out” these features as they arrive. This is described in “Extracting Information from HTML Data” on page 541.

You can either process the data as a stream or you can use the InputData class or object Sun WorkShop Visual provides to access it through the `getData()` and `getSize()` methods. This is particularly useful if you are downloading data to send to a display widget - for example, a gif or jpeg image. For C code, the InputData object is a data structure. For C++ and Java it is a class. See the online reference material for details of InputData by opening this file in an HTML browser and following the appropriate links:

```
$VISUROOT/lib/locale/<YourLocale>/sc/index.html
```

where VISUROOT is the install directory of your Sun WorkShop Visual and `<YourLocale>` is the locale you are using.

For a simple example of how to process incoming data, generate code from your design after setting up an Internet Smart Code callback with the “Go Live” toggle set which uses the “@<widgetname>” shorthand notation as the Receive Handler. This is exactly what you will do in “Simple Internet Smart Code Tutorial” on page 536 below.

Communication Protocol

Sun WorkShop Visual assumes, if you have chosen “Internet” Smart Code, that you are fetching data from a location on the Internet and therefore uses the GET HTTP protocol. If you override the send handler by specifying a function name for it in the Customize dialog, Sun WorkShop Visual uses the POST protocol.

Simple Internet Smart Code Tutorial

This example introduces you simply and quickly to Internet Smart Code. It connects to a real Web site and downloads data from it. You will see this happening both within Sun WorkShop Visual, using “Go Live”, and in your generated application.

In order to get you familiar with the use of Internet Smart Code quickly, this example does not attempt to parse the returned data. Parsing of HTML is described in “Extracting Information from HTML Data” on page 541.

Note – This tutorial shows you how to connect to a remote server, so make sure you are working from a computer configured to do this.

---

1. **Create a hierarchy containing the widgets shown in Figure 18-2.**

   These are: application shell->form->{button, scrolled text}.

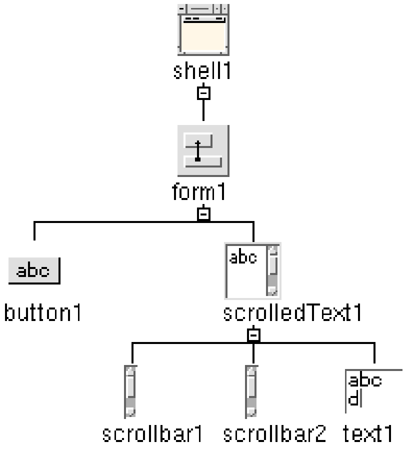


FIGURE 18-2 Hierarchy for Internet Tutorial

2. **In the Layout Editor, attach the scrolled text widget to the bottom and right edges of the form so that it resizes when the window is resized.**

   This is a purely cosmetic step - so that you are able to see the returned data more easily.

3. **Select text1 (the text area of the scrolled text widget). Press the “Add to New Group” on the toolbar.**

   This button is shown in Figure 18-3.

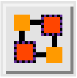
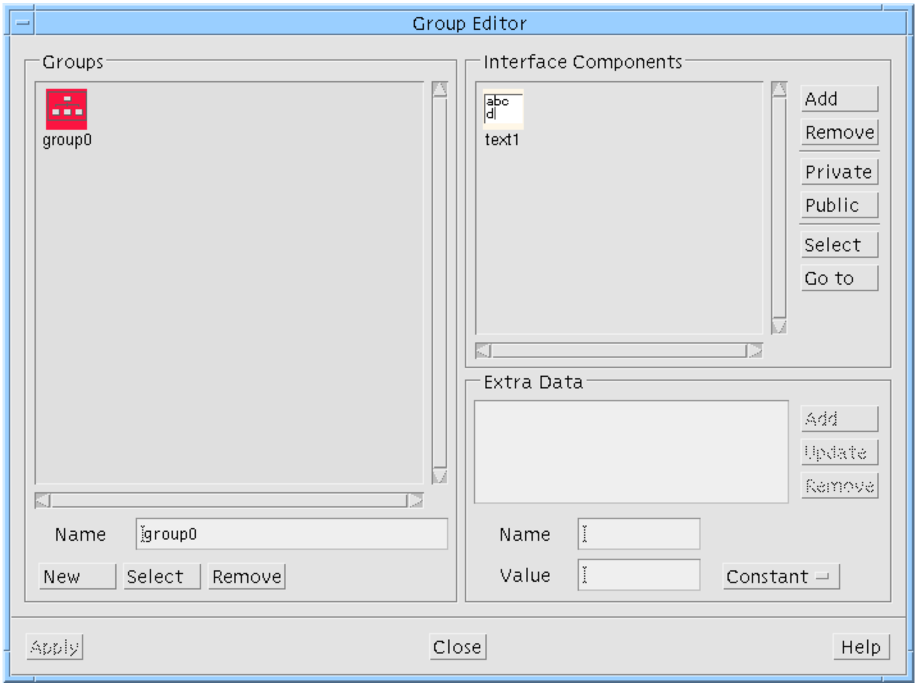


FIGURE 18-3 Add to New Group Toolbar Button

4. **When the Group Editor appears, check that it shows a group named Group0 containing a text widget as its only member, as shown in Figure 18-4.**



Name group0

FIGURE 18-4 Group Editor

5. **Close the Group Editor.**

   We do not need to make any changes, we shall use the Group as created by Sun WorkShop Visual.

6. **Select button1 and display the Callbacks dialog.**

7. **Check that "Activate" is selected from the list on the left and put in `goInternet` as the name of the callback.**

   Do not add this callback yet as we have to define the Smart Code for it.

8. **Set the "Smart Code" toggle.**

9. **Choose "Internet" form the option menu of Smart Code flavors.**

10. **Select Group0 as the Group for this callback.**

    Do this by pressing the "Group" toggle, making sure that Group0 is selected and pressing "Apply".

11. **Press the "Customize" button.**

    This displays the Customize dialog.

12. **In the Customize dialog, type the following URL in the URL field:**

```
http://www.ist.co.uk/index.html
```

13. **If you are behind a firewall, set the Proxy host and port.**

Tip – See “More on Proxies” on page 516 for more information on setting your proxies.

---

14. **In the Receive handler field put the following:**

```
@text1
```

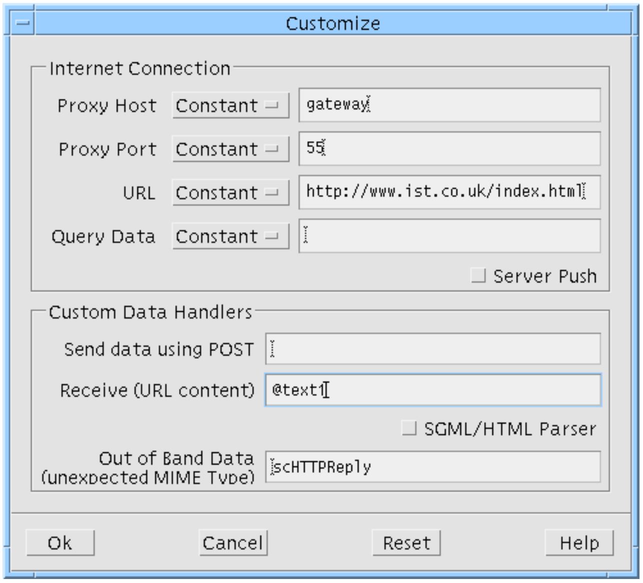
Tip – See “Going Live” on page 521 for information on the use of “@” in these fields.

---

15. **Press “Ok” in the Customize dialog.**

The completed Customize dialog is shown in Figure 18-5.

Note – The Customize dialog shows fictitious proxies as an example - you must enter those which are relevant to your network, as described in “More on Proxies” on page 516.



Customize

Internet Connection

Proxy Host: Constant — gateway

Proxy Port: Constant — 55

URL: Constant — http://www.ist.co.uk/index.html

Query Data: Constant —

□ Server Push

Custom Data Handlers

Send data using POST:

Receive (URL content): @text1

□ SGML/HTML Parser

Out of Band Data (unexpected MIME Type): scHTTPReply

FIGURE 18-5 Completed Customize Dialog

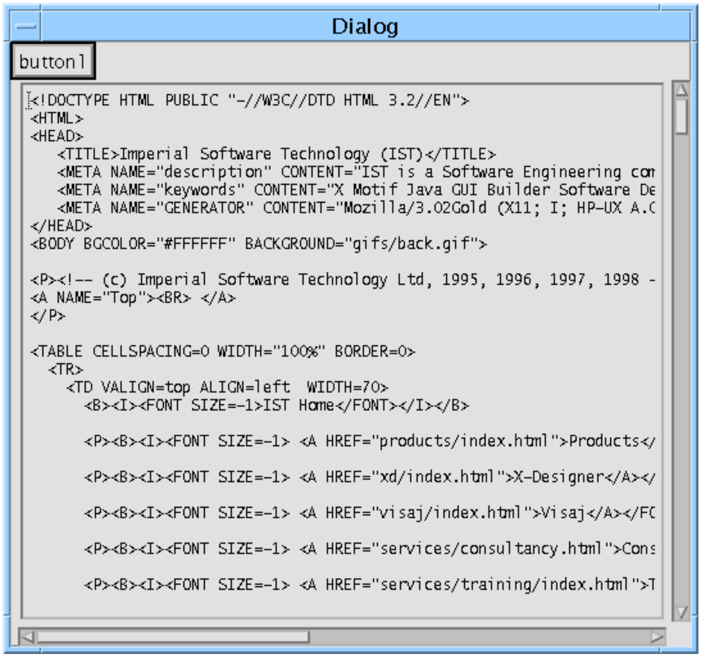
16. **Press “Add” to add your new callback.**

17. **Still in the Callbacks dialog, set the “Go Live” toggle.**

    When you set “Go Live” you do not need to “Update” the callback. The callback is immediate “Live”.

18. **In the dynamic display, press button1.**

    There is a a pause while the connection is made with the remote server, then the returned data (which is the Web page specified in the Customize dialog) appears in text1, as shown in Figure 18-6.



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
   <TITLE>Imperial Software Technology (IST)</TITLE>
   <META NAME="description" CONTENT="IST is a Software Engineering com
   <META NAME="keywords" CONTENT="X Motif Java GUI Builder Software De
   <META NAME="GENERATOR" CONTENT="Mozilla/3.02Gold (X11; I; HP-UX A.C
</HEAD>
<BODY BGCOLOR="#FFFFFF" BACKGROUND="gifs/back.gif">

<P><!-- (c) Imperial Software Technology Ltd, 1995, 1996, 1997, 1998 -
<A NAME="Top"><BR> </A>
</P>

<TABLE CELLSPACING=0 WIDTH="100%" BORDER=0>
  <TR>
    <TD VALIGN=top ALIGN=left  WIDTH=70>
      <B><I><FONT SIZE=-1>IST Home</FONT></I></B>

      <P><B><I><FONT SIZE=-1> <A HREF="products/index.html">Products</

      <P><B><I><FONT SIZE=-1> <A HREF="xd/index.html">X-Designer</A></

      <P><B><I><FONT SIZE=-1> <A HREF="visaj/index.html">Visaj</A></FC

      <P><B><I><FONT SIZE=-1> <A HREF="services/consultancy.html">Cons

      <P><B><I><FONT SIZE=-1> <A HREF="services/training/index.html">T
```

FIGURE 18-6 Live Dynamic Display

The final stage of this tutorial shows you the same occurring in the generated application.

19. **Generate code for your Internet enabled design.**

    You can generate any flavor of code, as long as you are able to compile it.

20. **Compile the generated code.**

21. **Run your client application.**

    Your application connects to the remote server and displays the specified Web page.

Going a Step Further

Having completed this tutorial, you may like to try some more advanced features of thin client and Internet Smart Code. Provided as part of your Sun WorkShop Visual package are HTML files containing instructions for running supplied Sun WorkShop Visual Replay scripts which run the tutorials for you. You simply watch it running and then examine the results. Open the following files in an HTML browser:

1. `$VISUROOT/lib/locale/<YourLocale>/sc/timex.html`. This describes the "Server Push" tutorial, demonstrating how to create an application with automatic remote update.

2. `$VISUROOT/lib/locale/<YourLocale>/sc/parsex.html`. This describes how to create an application which fetches a Web page and then parses it.

Note – VISUROOT is the install directory of your Sun WorkShop Visual and "YourLocale" is the name of the locale you are using.

Extracting Information from HTML Data

An application developed with Internet Smart Code might be used to fetch a Web page, parse it and display the result. To help you organize any HTML data returned by a server and to considerably simplify the process, a full, yet simple to use, HTML parser is supplied with Sun WorkShop Visual.

As a result of the origins and intentions of the World Wide Web, most of the data fetched from Web servers will be in HTML. The parser is based on the reference SGML parser materials from the SGML User Group[1]. It has been adapted to produce a general purpose SGML parser engine. SGML works in conjunction with a DTD (Document Type Definition) to define a markup language. The DTD for HTML is supplied with Sun WorkShop Visual

By adding extra DTDs, you can use the parser with other standard and in-house markup languages. You will also be able to upgrade your application for future versions of HTML and for XML.

The SGML parser has a simple and convenient programming interface. You register your interest in one or more features of the input stream (i.e. the HTML tags) and a routine of your choosing is called whenever the parser finds one of these features. This is analogous to the widget callback mechanism - widgets register their interest in certain actions and a given routine is called when such actions occur.

[1] Standard Generalized Markup Language Users' Group (SGMLUG) SGML Parser Materials. Written by James Clark.

Note – If you are not familiar with the Web technology which this uses (or you are confused by the list of acronyms), you may need to do some background reading. See "Books on Networking and World Wide Web" on page 888, for a list of suggested books.

To tell Sun WorkShop Visual that you wish to use the parser to extract key information from the incoming data, set the "SGML / HTML Parsing" toggle in the Customize dialog. In your Receive Handler, set up the parser according to your requirements and then send the data to the parser. Exactly how to do all of this (and what happens next) is detailed below.

Using the Parser

Once you have told Sun WorkShop Visual that you wish to process SGML / HTML by selecting the toggle in the Customize dialog, the following four steps are needed. Each of these takes place inside your Receive Handler:

1. Register the MIME type by calling the routine `scRegisterSGMLMimeType` (or the shortcut for HTML `scRegisterHTML`).
2. Register an error handler by calling the routine `scRegisterSGMLMimeErrorHandler`. This is an optional step.
3. Register an interest in one or more features of the input stream by calling the routines `scAddTagCallback` and `scAddAttrCallback`. Alternatively at this point you can request a traditional parse tree.
4. Call the parser using the routine `scProcessSGML`.

Each of these steps is examined more closely in the following sub-sections.

Before programming the interface to the parser, make sure that you are including the following header file:

```
#include <SGML.h>
```

The directory of this header file, which is part of the Sun WorkShop Visual distribution, is automatically included in the Makefile.

In addition, you will need to set the DTDDIR environment variable to:

```
$VISUROOT/src/sgml/dtds
```

"Practical Information for Using the Parser" on page 551 provides some more information on the location of the SGML parser and the files it uses.

Registering the MIME Type

In order to configure the parser, you first need to create an SGML object. This object is then passed to any other routines you need to call. An SGML object is returned from the routine you call to register the MIME type of your data, which is shown below:

```
SGML_t *
scRegisterSGMLMimeType( mimetype, dtd)
        char * mimetype;
        char * dtd;
```

Use this to associate a MIME type with an SGML DTD. The most common will be:

```
SGML_t * sgm = scRegisterSGMLMimeType( “text/html”, “HTML32.soc”);
```

Because this is the most commonly used, the following is supplied as a shortcut:

```
SGML_t *
scRegisterHTML( mimetype)
        char * mimetype;
```

This does exactly the same as the one above, associating “text/html” with the HTML32 DTD. Add your own DTD by placing it in the directory referenced by the DTDDIR environment variable.

The following shows what is generated when “processMyData” has been specified as the Receive Handler, with “SGML/HTML Parser” set. A line has been added to register the MIME type:

```
int
processMyData ( data, idata)
        sc_stdcs_t* data;
        sc_idata* idata;
{
    extern InputData * newInputData();

    group0_t * group = (group0_t*)data->group;
    InputStream * i   = (*idata->getInputStream)( idata);
#if 0 /* example usage */
    char      * type  = (*idata->getMimeType)(idata);
    int         len   = (*idata->getContentLength)( idata);

    InputData * id    = newInputData( i);
    char *   d        = (*id->getData)( id);
```

This line has been added →

```
#endif
    sgm = scRegisterHTML( type);
    ...
    return 0;
}
```

Registering an Error Handler

The default error handler outputs error messages to standard output. You can override this by registering your own error handler using the following routine:

```
int
scRegisterSGMLErrorHandler( void_f errorhandler)
```

Your error handler should be of the form:

```
void
errorhandler( char * s)
```

Registering Interest in Input Stream Features

To access the parsed data, you should register an interest in one or more features of the input (e.g. particular tags and attributes in HTML).

Registering interest in input features is directly analogous to the widget callback mechanism where widgets register their interest in certain actions and a specified routine (callback) is called when the action occurs. Here, you register your interest in features of the language and the parser calls your callback routines when it comes across one of these features.

There are two major features of HTML: tags and attributes. You can register an interest in these features using one of the two routines described in subsequent sections. First, though, a brief description of what is meant by *tag* and *attribute*.

Tags

Tags are features of HTML which describe the format of the following piece of text. Tags appear in angle brackets, for example <menu> to indicate a bulleted list or <code> to indicate a code listing. The following example shows a “menu” block containing individual list items:

```
<menu>
    <li>The first item in the list</li>
    <li>The second item in the list</li>
</menu>
```

Attributes

Attributes are another feature of HTML. They also appear inside angle brackets. Attributes are placed after the tag and are used to indicate a reference. This may be an external file, an image or a position elsewhere in the document. Attributes are always made up of a reserved string, an equals sign (=) and the reference. The following example shows two attributes. The first, an “href”, names the destination of a link (somewhere called “bottom”). The text inside the block, “Go to bottom of page”, is the “visible” part of this link. The second attribute is a “name”. It names a location - in this case “bottom”. So, from a user point of view, selecting “Go to bottom of page”, moves the view to the named location “bottom”:

```
<a href=”#bottom”><b>Go to bottom of page</b></a>
...
<a name=”bottom”></a>
```

There are two routines which you can use to register your interest in particular HTML language features. These are:

1. scAddTagCallback. Use this to register an interest in a particular tag.
2. scAddAttrCallback. Use this to register an interest in an attribute. This is the same as scAddTagCallback but configured to show an interest only in attributes.

The following sections explain these registration routines.

Registering Interest in Tags

The SGML parser needs to know which parts of the HTML input you are interested in. It also needs to know *at what point* within the selected block of HTML to call your callback routine.

The routine for registering interest in tag elements is:

```
int
scAddTagCallback( SGML_t * sgm, char * tagname, int type, void
                                   (*callback)(), void * data)
```

The parameters to this routine need more explanation. They are detailed in the following sub-sections.

SGML_t * sgm

This is the SGML object returned from the `scRegisterSGMLMimeType`, the routine for registering the MIME type of your data. `scRegisterSGMLMimeType` is described in “Registering the MIME Type” on page 543.

char * tagname

This is the tag in which you are interested. Do not include the angle brackets, simply the tag itself in upper case e.g. “A” or “MENU” or “LI”.

int type

This parameter tells the parser *when* within the chosen tag to call your callback routine. In addition, which “type” you choose determines whether your callback routine is passed any of the data from inside the selected block of HTML or not. Your callback routine always receives the tagname and the type (so that you can use the same routine for any number of different tags and types) but only “ON_ATTR” and “ON_DATA” cause any more information to be returned.

You have a choice of four pre-defined types, according to where in the tag block you wish your callback routine to be called as illustrated in Figure 18-7. The four types are:

- ON_ENTRY. This refers to the beginning of a block (e.g. <a> or <menu>). No data (referred to as *call_data* in “Your Callback Routine” on page 547) is passed to your routine.
- ON_EXIT. This refers to the end of a block (e.g. </a> or </menu>). No data (referred to as *call_data* in “Your Callback Routine” on page 547) is passed to your routine.
- ON_ATTR. This refers to the attribute appearing inside a tag definition (e.g. href=”mylink”). Your callback routine receives the text inside the quotation marks as its call_data parameter (as explained in “Your Callback Routine” on page 547).
- ON_DATA. This refers to the text (or data) after the tag. Your callback routine receives the text between the beginning and the end of the tag as its call_data parameter (as explained in “Your Callback Routine” on page 547).

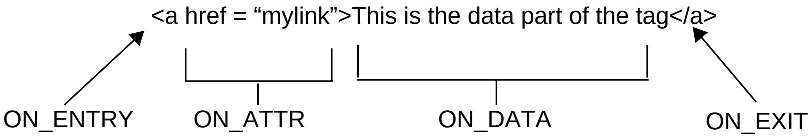


FIGURE 18-7 Types

void (*callback)()

This is the name of the routine which the parser should call when it comes across a tag you are interested in (the callback routine). This is a routine defined by you. The format of this routine is described in “Your Callback Routine” on page 547.

void * data

This parameter gives you a chance to pass data to your callback routine. This will be passed to your routine as its “client data” parameter.

You may register an interest in any number of tags, calling this routine once for each tag.

Registering Interest in Attributes

The routine for registering interest in attributes is:

```
int
scAddAttrCallback( sgm, tagname, attrname, callback, data)
    SGML_t * sgm;
    char * tagname;
    char * attrname;
    void (*callback)();
    void * data;
```

The only parameter which is different from those described for the tag registering routine above, is:

- `char * attrname`. This is the name of the attribute in which you are interested.

You may register an interest in any number of attributes.

Your Callback Routine

There is no stub file for your callback routine. You must write it all yourself. The name of the routine is the name specified as the fourth parameter to `scAddAttrCallback` or `scAddTagCallback` (that is, the parameter called “callback”).

Your callback routine is called by the parser when it detects a tag or attribute in which you have registered an interest. The following example shows what your callback should look like and lists the parameters passed in:

```
int
mycallback( tag, attribute, type, call_data, client_data)
        char * tag;
        char * attribute;
        int    type;
        void * call_data;
        void * client_data;
```

The parameters passed into your routine are:

1. `char * tag`. This is the tag which the parser detected.
2. `char * attribute`. This is the attribute which the parser detected. If you were only interested in tags, this is null.
3. `int type`. This is whether the routine has been called “ON_ENTRY”, “ON_EXIT”, “ON_DATA” or “ON_ATTR”. These four are discussed in “int type” on page 546. This parameter is not relevant when you are interested in attributes.
4. `void * call_data`. When you are interested in attributes, this is the part of the attribute which comes after the equals sign. For example, if you have registered an interest in the “href” attribute and the parser finds the following line:

   ```
   <a href=”#regmime”>
   ```

then this parameter would be “`#regmime`”.

If you have specified “ON_DATA” as the type, this gives you data after the tag. See Figure 18-7 on page 546 for an illustration.

5. `void * client_data`. This is the “data” parameter passed to the registration routine, allowing you to pass your own data into the callback. This is similar to the client data for Xt callbacks, as seen in the Callbacks dialog.

Because you may register an interest in any number of tags or attributes, you can also have any number of callback routines, but having one for tags and another for attributes is probably the most useful combination.

Parsing the Input Stream

Once you have configured an SGML object by specifying the MIME type, registering error handlers and registering interest in particular input stream features, you are ready to call the parser. Here is the routine to do this:

```
int
scProcessSGML( sgm, istream)
     SGML_t * sgm;/* the parser handle scRegisterSGMLMimeType */
     InputStream * istream;/* the input stream from the server */
```

The first parameter is described in the preceding sections. The second parameter, the input stream, is passed to your Receive Handler.

Using the Parser - Example

This section provides an illustration of the use of the SGML parser. When you specify in the Customize dialog that you wish to have SGML parsing, you also need to provide a name for a Receive Handler. Here is the stub for the handler which is generated by Sun WorkShop Visual:

```
int
processMyData ( data, idata)
        sc_stdcs_t* data;
        sc_idata* idata;
{
        extern InputData * newInputData();

        group0_t * group = (group0_t*)data->group;
        InputStream * i   = (*idata->getInputStream)( idata);
#if 0 /* example usage */
        char      * type  = (*idata->getMimeType)(idata);
        int         len   = (*idata->getContentLength)( idata);

        InputData * id    = newInputData( i);
        char *   d        = (*id->getData)( id);
#endif
        return 0;
}
```

In order to make use of the SGML parser, you will have to add some code to this routine in order to create an SGML object, configure it and then send the incoming data to the parser. Here is the Receive Handler with extra code for parsing incoming HTML:

```
int
```

```
processMyData ( data, idata)
        sc_stdcs_t* data;
        sc_idata* idata;
{
    extern InputData * newInputData();

    group0_t * group = (group0_t*)data->group;
    InputStream * i   = (*idata->getInputStream)( idata);
```

This line has been removed

```
~~#if 0 /* example usage */~~
    char      * type  = (*idata->getMimeType)(idata);
    int         len   = (*idata->getContentLength)( idata);

    InputData * id    = newInputData( i);
    char *   d        = (*id->getData)( id);
```

This line has been removed

```
~~#endif~~

    SGML_t * sgm;
```

These lines have been added

```
    if ( strcmp( type, “text/html”) != 0)
          return -1;
    sgm = scRegisterHTML( type); /* the parser object */
    (void) scAddTagCallback(sgm, “A”, ON_ENTRY, getanchor,
                 “a-call”);
    (void) scAddAttrCallback(sgm,  “A”, “HREF”,
                 getlinkinfo, “href”);
    (void) scProcessSGML( sgm, i);

    return 0;
}
```

This routine specifies that `getanchor` should be called whenever the parser finds an anchor tag (<a>) and `getlinkinfo` should be called whenever the “href” attribute is found. These routines, written by yourself, should look like this:

```
int
getanchor( tag, attr, type, call_data, client_data)
    char * tag;
    char * attr;
    int    type;
    void * call_data;
    void * client_data;
{
    printf(“anchor-start(%s)\n”, client_data);
}

int
getlinkinfo( tag, attr, type, call_data, client_data)
    char * tag;
    char * attr;
    int    type;
    void * call_data;
    void * client_data;
{
    printf( “%s=%s\n”, client_data, call_data);
}
```

Note – See “Going a Step Further” on page 541 for information on how to run an on-line Sun WorkShop Visual Replay script which makes use of the parser.

Practical Information for Using the Parser

To use the SGML parser, you will need to link with precompiled code. The sources are available in:

`$VISUROOT/src/sgml`

The license provisions mean that you are free to use them as you wish.

To begin with, it is easier to use the precompiled version which comes with Sun WorkShop Visual.

The SGML parser uses the following files and directories:

1. `$VISUROOT/lib`. This contains an archive and a shared version of the SGML library. The make rules, generated by Sun WorkShop Visual into the `Makefile`, use `libsgml.so`, but you can link with `libsgml.a` if you prefer.
2. `$VISUROOT/src/sgml/hdrs/SGML.h`. This is the include file necessary to use the parser engine API. is referenced in the Makefile if you set the “SGML/HTML parsing” toggle in the Smart Code Customize dialog.

3. `$VISUROOT/src/sgml/dtds`. This is the directory containing the HTML 3.2 DTD and other related data files. The parser will need to find this, so you need to set the DTDDIR environment to:

   ```
   $VISUROOT/src/sgml/dtds
   ```

Before compiling, you should make sure that:

1. `$VISUROOT/bin` is in your PATH
2. `$VISUROOT/lib` has been added to your library path environment variable (for example LD_LIBRARY_PATH for 32 bit applications and LD_LIBRARY_PATH64 for 64 bit applications.).

CHAPTER 19

Makefile Generation

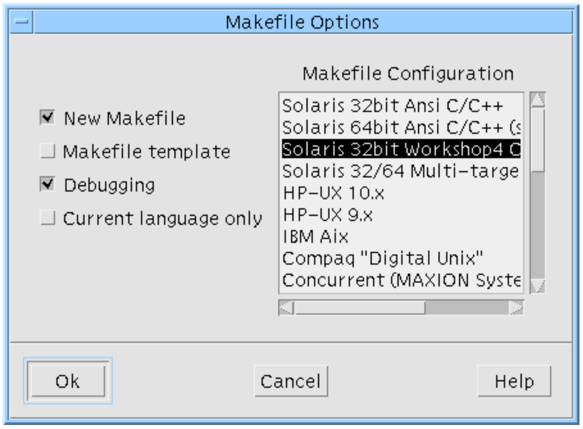
Introduction

This section describes Sun WorkShop Visual's Makefile generation facilities. Sun WorkShop Visual can create two types of Makefile: simple or with templates. The simple Makefile only contains the build rules for the local *.xd* file and so is only useful for applications that are contained in a single file. A Makefile with templates can be updated to add files without overwriting your previous work. Unlike most generated files, Makefiles with templates can be edited and regenerated without losing your work.

This section describes the Makefile options available when you generate code for a design. It also provides a short tutorial with step-by-step instructions for creating a Makefile with templates and then updating the Makefile when a second design file is added to the application. Following this tutorial enables you to familiarize yourself with Sun WorkShop Visual's Makefile generation capabilities.

Makefile Generation Options

Pressing the button labelled “Options” beside the makefile text box produces the dialog shown in Figure 19-1.



Makefile Options

☑ New Makefile

☐ Makefile template

☑ Debugging

☐ Current language only

Makefile Configuration

- Solaris 32bit Ansi C/C++
- Solaris 64bit Ansi C/C++ (s
- Solaris 32bit Workshop4 C
- Solaris 32/64 Multi-targe
- HP-UX 10.x
- HP-UX 9.x
- IBM Aix
- Compaq "Digital Unix"
- Concurrent (MAXION Syste

FIGURE 19-1 Makefile Options Dialog

The dialog contains four toggles labelled “New Makefile”, “Makefile template” and “Debugging” and a scrolled list. Setting “Debugging” simply adds the “-g” flag to the list of flags to be sent to the compiler so that you can build a version of your application for debugging.

When you generate code in one language and then generate another set of code files in another language, all Makefiles generated afterwards will contain rules for both sets of generated files. Setting the “Current language only” toggle ensures that you are generating a Makefile for the current language only.

New Makefile and Makefile Template Toggles

“New Makefile” and “Makefile Template” relate to the two different types of makefile that you can generate: a simple makefile and a makefile with templates. Which one is generated depends on the way in which the toggles in this dialog are set. The two toggles work in conjunction with one another. There are four ways they can be set:

1. The “New Makefile” toggle is set and the “Makefile Template” toggle is not set.

   This generates a simple Makefile. You would use this option if you do not wish to add other design files to the application which the makefile will build.

2. Both toggles are set.

   This generates a makefile with templates - i.e. with structured comments which serve as templates for updating the file on subsequent generations. Use this option if you are going to add other design files to the application which the makefile will build.

3. The “New Makefile” toggle is not set and the “Makefile Template” toggle is set.

   This will add the current design code files to the Makefile which was generated as described in number 2 above.

4. Neither toggle is set.

   This works in exactly the same way as number 3 above.

The List of Makefile Types

The scrolled list in the Makefile Options dialog allows you to generate a Makefile for a specific target platform. For some platforms more than one option is available, allowing for different environments. On Solaris, for example, you can choose compile your application for the 32-bit or the 64-bit architectures. The default for your platform is selected initially. Change this if you wish to generate a Makefile to build your application on another platform or if you wish to use a different compiler from the default. You can, of course, generate any number of different Makefiles to match the environments in which your application will be running.

In some cases, where there is more than one option for a given platform, you can generate a “multi-target” Makefile. This allows you to specify any of the available targets using the one Makefile. For example, the following selection from the Makefile Options dialog:

```
Solaris 32/64 Multi-target C/C++ (sparc)
```

generates a Makefile capable of doing the same as all the other Solaris Makefiles available from the Makefile Options dialog. These are:

```
Solaris 32bit Ansi C/C++
Solaris 64bit Ansi C/C++ (sparc)
Solaris 32bit Workshop4 Compatible C++
```

Multi-target Makefiles are identified in the Makefile Options dialog by the words “Multi-target”. Look at the Makefile itself for the list of command-line options available.

If you wish to build just one version of your application, choose the appropriate Makefile type. If you intend to build your application anumber of times - once for each different target on a given platform, you may find the “multi-target” Makefile most useful.

Makefile Generation Notes

There are a few points to bear in mind when generating a makefile:

- If you are going to use the makefile with templates, make sure that you create the first makefile (the one with both “New Makefile” and “Makefile Template” toggles set) for the main design - usually the one which contains the application shell. Sun WorkShop Visual will assume that the name of the application is the same as the name of the primary source file of the design from which a “New Makefile” is generated.
- When you choose “Makefile Template” only, make sure that you have specified the name of the makefile which has already been generated. Sun WorkShop Visual will report an error if you give another name.
- You can edit a makefile with templates and regenerate it without losing your changes. This is not the case for simple makefiles.

Creating the Initial Makefile

The first step is to create a design, generate the C code for it and then generate an initial Makefile. It is assumed that you are familiar with the general use of Sun WorkShop Visual.

1. **Create a new directory** ***myapp*****. Make this your current directory and start Sun WorkShop Visual.**
2. **Build the widget hierarchy shown in Figure 19-2**

   This is an Application Shell with a Form containing a Button.

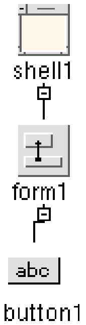


FIGURE 19-2 Main Program Widget Hierarchy

3. **Give the PushButton an Activate callback,** *button_pressed*.

   Before you generate a Makefile, you must generate the code files that you want to include in it. Generating the code files sets the names for these files in the design file. Until you do this, the Makefile generation feature doesn't know the names of these files and can't add them to the Makefile.

4. **Display the Generate dialog and make sure that "C" is the selected language.**

   The Generate dialog will be primed with filenames based on the save file name or "untitled" if the design has not been saved.

5. **Type: `myapp.c` into the "Code" field and set the "Generate" toggle.**

6. **Type: `myapp.c` into the "Main Program" field and set the "Generate" toggle.**

   If the "code" and "Main Program" filenames are the same, the code file is generated with a main procedure.

7. **In the Code Options dialog, set the "Include Header File" toggle.**

8. **Type: `app_stubs.c` into the "Stubs" field and set the "Generate" toggle.**

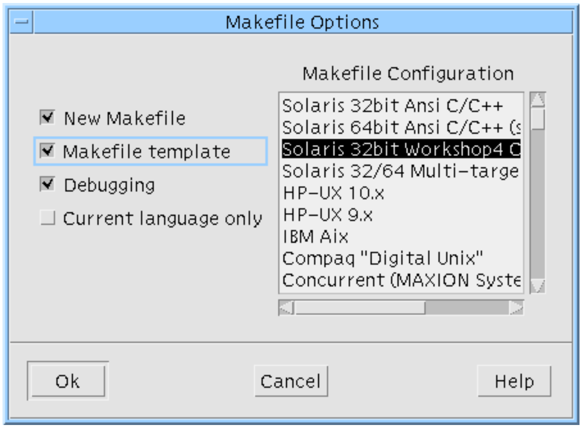
9. **In the Generate Options dialog, set the "Links" option menu to "None".**

   Now you can generate a Makefile that compiles and links *myapp.c* and *app_stubs.c*:

10. **Type: `Makefile` into the "Makefile" field and set the "Generate" toggle.**

11. **In the Makefile Options dialog set both "New Makefile" and "Makefile Template" toggles on, as shown in Figure 19-3.**

    The scrolled list on the right of the dialog is described in "The List of Makefile Types" on page 555.



Makefile Options

☑ New Makefile
☑ Makefile template
☑ Debugging
☐ Current language only

Makefile Configuration

Solaris 32bit Ansi C/C++
Solaris 64bit Ansi C/C++ (s
Solaris 32bit Workshop4 C
Solaris 32/64 Multi-targe
HP-UX 10.x
HP-UX 9.x
IBM Aix
Compaq "Digital Unix"
Concurrent (MAXION Syste

FIGURE 19-3 Initial Makefile Generation

12. **In the Generate dialog press the “Generate” button.**

    The generated Makefile contains the required make rules and template lines for further amendment. Ignoring the template lines, the generated Makefile contains the following rules:

```
XD_C_PROGRAMS=\
          myapp
XD_C_PROGRAM_OBJECTS=\
          myapp.o
XD_C_PROGRAM_SOURCES=\
          myapp.c
XD_C_STUB_OBJECTS=\
          app_stubs.o
XD_C_STUB_SOURCES=\
          app_stubs.c
myapp: myapp.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)
    $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp myapp.o
$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
myapp.o: myapp.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c myapp.c
app_stubs.o: app_stubs.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c app_stubs.c
```

13. **Save the current design to** ***myapp.xd*****.**

Updating the Initial Makefile

Once you have generated the initial Makefile, you can update it to reflect additional work. To demonstrate this, use the following instructions to build a popup dialog in another file, generate code for it and then update the Makefile to reflect the new modules.

1. **Select “New” from the File menu to start a new design.**
2. **Build the hierarchy shown in Figure 19-4.**

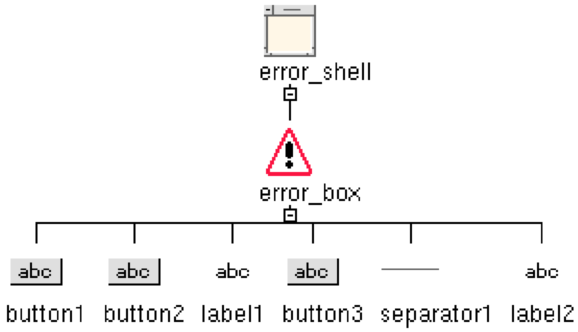


FIGURE 19-4 Secondary Popup Dialog

3. **Name the Shell and MessageBox** *error_shell* **and** *error_box* **respectively.**

4. **Give the MessageBox a Cancel callback,** *cancel_error***.**

5. **Display the Generate dialog and make sure that "C" is the selected language.**

6. **Type: `error.h` into the "Externs" field and set the "Generate" toggle.**

7. **Type: `error.c` into the "Code" field and set the "Generate" toggle.**

8. **In the Code Options dialog, set the "Include Header File" toggle and type: `error.h` into the corresponding field.**

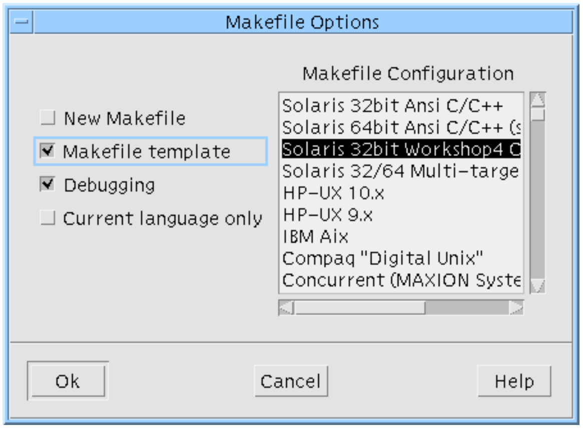
9. **Type: `error_stubs.c` into the "Stubs" field and set the "Generate" toggle.**

10. **Make sure that the "Generate" toggle next to "Main Program" is not set.**

11. **In the Generate Options dialog, select "None" from the "Links" option menu.**

12. **Type: `Makefile` into the "Makefile" field and set the "Generate" toggle.**

13. **In the Makefile Options dialog turn off the "New Makefile" toggle, leaving the "Makefile Template" toggle set, as shown in Figure 19-5.**



Makefile Options

Makefile Configuration

□ New Makefile
☑ Makefile template
☑ Debugging
□ Current language only

Solaris 32bit Ansi C/C++
Solaris 64bit Ansi C/C++ (s
Solaris 32bit Workshop4 C
Solaris 32/64 Multi-targe
HP-UX 10.x
HP-UX 9.x
IBM Aix
Compaq "Digital Unix"
Concurrent (MAXION Syste

FIGURE 19-5 Updating Makefile

14. Press the "Generate" button in the Generate dialog.

The generated Makefile is updated with the new modules.

```
XD_C_PROGRAMS=\
myapp

XD_C_PROGRAM_OBJECTS=\
myapp.o

XD_C_PROGRAM_SOURCES=\
myapp.c

XD_C_OBJECTS=\
error.o

XD_C_SOURCES=\
error.c

XD_C_STUB_OBJECTS=\
error_stubs.o\
app_stubs.o

XD_C_STUB_SOURCES=\
error_stubs.c\
app_stubs.c

myapp: myapp.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)
$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp\
myapp.o $(XD_C_OBJECTS)\
$(XD_C_STUB_OBJECTS) $(MOTIFLIBS)\
$(LDLIBS)

myapp.o: myapp.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c myapp.c

error.o: error.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c error.c
```

```
app_stubs.o: app_stubs.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c app_stubs.c

error_stubs.o: error_stubs.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c error_stubs.c
```

Building the Application

Sun WorkShop Visual has generated the code files for two dialogs, two stub files and a Makefile. All that remains is to fill in the two stubs and build the application.

1. **Edit *app_stubs.c* and make the following changes.**

   At the end of the generated list of includes, add the following line:

   ```
   #include <error.h>
   ```

   Add functionality to the *button_pressed* callback stub. This callback pops up the error dialog when the button is pressed:

   ```
   void
   button_pressed (Widget w, XtPointer client_data, XtPointer
                                          call_data )
   {
       ...
       if ( error_shell == NULL )
            create_error_shell (XtParent (XtParent (w) ) );
       XtManageChild ( error_box );
   }
   ```

2. **Edit *error_stubs.c* and make the following changes.**

   Add functionality to the *cancel_error* callback. This function pops down the error dialog when the user presses the “Cancel” button.

   ```
   void
   Widget w, XtPointer client_data, XtPointer{
       ...
       XtUnmanageChild ( error_box );
   }
   ```

3. **Save the current design to *error.xd*.**

4. **Open the file *myapp.xd* in Sun WorkShop Visual.**

5. **Generate X resources into the file *myapp.res*.**

6. **To access these resources do the following:**

7. **If you are using a C shell enter on the command line:**

   ```
   setenv XENVIRONMENT myapp.res
   ```

8. **Or, if you are using a Bourne shell or ksh, enter on the command line:**

   ```
   XENVIRONMENT=myapp.res export XENVIRONMENT
   ```

9. **Set VISUROOT to the path of the Sun WorkShop Visual installation directory.**

   This must be done as the Makefile includes files and libraries relative to the Sun WorkShop Visual installation directory.

10. **Build the application. On the command line, type: `make`**

11. **To run the application, type: `myapp`**

Editing the Generated Makefile

You can edit and regenerate the Makefile without losing information. You can make the most commonly needed changes by editing the Makefile flags at the beginning of the file. For example, to make the compiler search the *../hdrs* directory for header files, append the entry:

```
-I../hdrs
```

to the end of the CFLAGS line in the Makefile.

The change to *CFLAGS* is retained when you regenerate the Makefile with the “New makefile” toggle off. It is only lost if you generate a new Makefile.

Editing Template Lines

A large part of the generated Makefile consists of *template lines*. Template lines are comments that control the generation of information into the Makefile. Template lines have a #Sun WorkShop Visual*:* prefix. For example, the following template lines tell Sun WorkShop Visual how to generate the Makefile lines that produce a C object file from a C source file (*XDG_C_SOURCE*):

```
#Sun WorkShop Visual:XDG_C_OBJECT: XDG_C_SOURCE

#Sun WorkShop Visual:    $(CC) $(CFLAGS) $(CPPFLAGS) -c XDG_C_SOURCE
```

Each time you update the Makefile to add a file to your application, Sun WorkShop Visual generates a *template instance* for each relevant template. These instances contain the actual build commands for your application.

Template instances are marked with a "DO NOT EDIT" comment at the beginning and at the end. A typical instance of the template shown above looks like:

```
#DO NOT EDIT >>>
error.o: error.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c error.c
#<<< DO NOT EDIT
```

Template instances should not be edited because your edits may be lost the next time you generate the Makefile. Instead, to change the build commands, edit the corresponding template lines. After you edit a template line, delete any instances of that template line that already exist in the Makefile. The instances are found just after the template line.

For example, to build all C files for debugging, you would:

1. **Change the template line:**

   ```
   #Sun WorkShop Visual: $(CC) $(CFLAGS) $(CPPFLAGS) -c XDG_C_SOURCE
   ```

   to

   ```
   #Sun WorkShop Visual: $(CC) $(CFLAGS) $(CPPFLAGS) -g -c XDG_C_SOURCE
   ```

2. **Remove the instances following the template line.**
3. **Regenerate the Makefile, with the "New" toggle off, for each design in the application.**

This procedure generates new instances using the modified template.

Template Configuration

The original templates are specified by the file pointed to by the following resources:

```
visu.motifMakeTemplateFile: $VISUROOT/make_templates/motif
visu.mmfcMakeTemplateFile: $VISUROOT/make_templates/mfc
```

There are two resources so that you can have different templates customized to pick up the appropriate class libraries. The value for the resource can contain environment variables which will be expanded by /bin/sh.

If Sun WorkShop Visual cannot find the file specified, it will fall back to the template specified in the Sun WorkShop Visual resource file, using the *makefileTemplate* application resource. To make a change to the template apply globally to all new Makefiles, edit the resource file. For details, see the *Configuration* chapter.

Sun WorkShop Visual refers to the template only when you generate a new Makefile. To change templates in an existing Makefile, edit the file by hand as described in the previous section, or delete the file and start over.

Dependency Information

Sun WorkShop Visual does not generate dependency information into the Makefile. The default template includes a "depend" target that can be invoked using:

```
make depend
```

This operation invokes the *makedepend* utility, which scans the existing makefile and appends a standard dependency list. Use *man makedepend* for more information.

Using Your Own Makefiles

The $VISUROOT/*make_templates* directory contains the template used to generate makefiles. This template contains the include directories and libraries you will need for each system configuration supported. The configuration chosen from the Makefile Options dialog determines which "block" in the template to use. There is always a default configuration for your system so you do not necessarily need to select one from this dialog. See "The List of Makefile Types" on page 555 for more details. A typical "block" is shown below:

```
##: system Solaris 32bit Ansi C/C++
##: default cpp
#UILFLAGS=-I${MOTIFHOME}/include/uil
#MRMLIBS=-L${MOTIF_LIB_DIR} -lMrm
#CPPFLAGS=-Ddrem=remainder -DS_SUNOS5
#MOTIFHOME=/usr/dt
#MOTIFINCLUDES=-I${MOTIFHOME}/include
#MOTIF_LIB_DIR=${MOTIFHOME}/lib${SYSDIR} -R${MOTIFHOME}/lib${SYSDIR}
#XINCLUDES=${MOTIFINCLUDES} -I/usr/openwin/include -I/usr/openwin/
include/X11
#X11_LIB_DIR=/usr/openwin/lib${SYSDIR} -R/usr/openwin/lib${SYSDIR}
#XSYSLIBS=-L${X11_LIB_DIR} -lXt -lX11 -lXext -lnsl -lsocket -lgen
#MOTIFLIB=-L${MOTIF_LIB_DIR} -lXm
#CC=cc
#CCC=CC
```

```
#ABI2CFLAGS=
#ABI2CCFLAGS=-compat=5
#ABI2LDFLAGS=
#ABI2ABIDIR=/ansi32
#ABI2SYSDIR=
#ABICFLAGS=${ABI2CFLAGS}
#ABICCFLAGS=${ABI2CCFLAGS}
#ABILDFLAGS=${ABI2LDFLAGS}
#SYSDIR=
#ABIDIR=/ansi32
##: end
```

XINCLUDES and *XLIBS* specify the extensions to the included directory and library path respectively.

CHAPTER 20

Advanced Layout

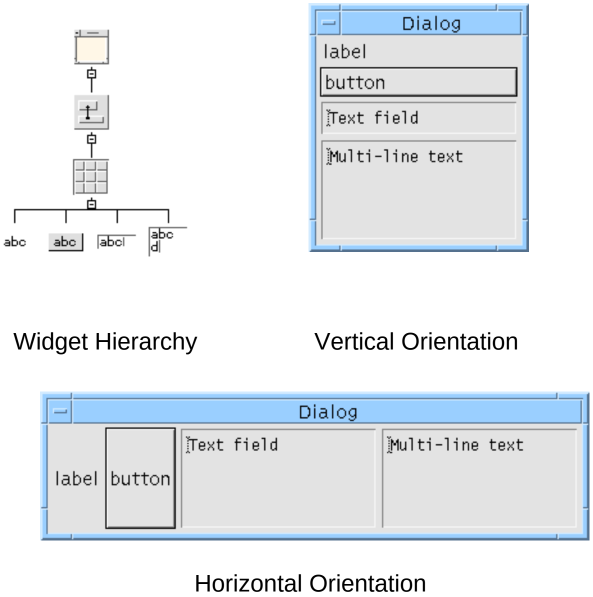
Introduction

This chapter describes some techniques that can be used to achieve more complicated widget layouts.

Column Layout Using the RowColumn

Dialog elements are frequently arranged in a column or row. This is easy when only a single column is needed but requires more work to create multiple columns.

The easiest way to create a single column layout is to use a RowColumn widget instead of a Form. The RowColumn can take almost any widget as a child and different widget types can be children of the same RowColumn. If the orientation of the RowColumn is vertical, it produces a single column; if horizontal, it produces a single row. These results are shown in Figure 20-1.



Widget Hierarchy

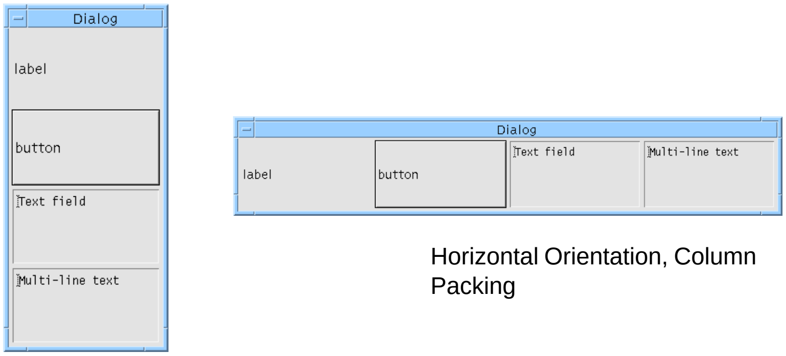
Vertical Orientation

Horizontal Orientation

FIGURE 20-1 Single-Column Layouts

Figure 20-1 shows the default behavior of the RowColumn widget when the Packing resource is set to “Tight”. “Tight” packing produces exactly one column or row. In the column layout, all widgets are constrained to the same width but they can have different heights; in the row layout, all widgets are constrained to the same height but they can have different widths.

If you set Packing to “Column”, both the width and height of all widgets are the same, as shown in Figure 20-2.



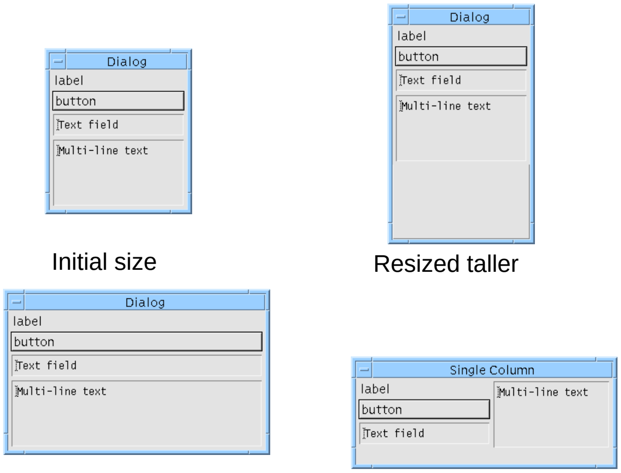
Horizontal Orientation, Column Packing

Vertical Orientation, Column Packing

FIGURE 20-2 Column Packing

Resize Behavior of RowColumns

Since any dialog can be potentially resized, the behavior of its components upon resizing is always important. There is no general documentation of resize behavior; you have to discover it by experimentation. A vertical RowColumn with “Tight” packing behaves as shown in Figure 20-3.



Initial size

Resized taller

Resized wider

Resized wider and shorter

FIGURE 20-3 RowColumn Resizing

All children in a RowColumn widget are displayed if at all possible. If the RowColumn is not large enough to display all of its children, the ones that don't fit are not displayed at all. It is rare to see only part of a child widget.

Multiple Columns

You can use the RowColumn widget to lay out widgets in multiple columns, as shown in Figure 20-4.

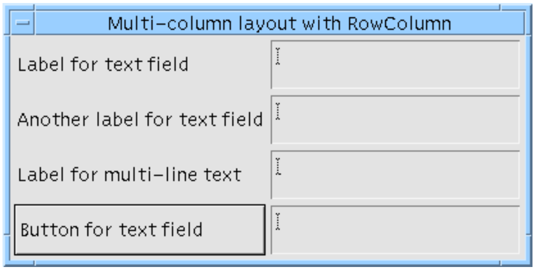

Multi–column layout with RowColumn
Label for text field
Another label for text field
Label for multi–line text
Button for text field

FIGURE 20-4 Multiple Column Layout with RowColumn Widget

This layout has limitations, however. In order to get more than one column, you must set Packing to "Column", which forces all of the children to be the same size. In this case, because one of the text boxes is 2 lines high, all of the text boxes must be that size. The result wastes space and may be confusing to the user. The layout shown in Figure 20-5 is an improvement:

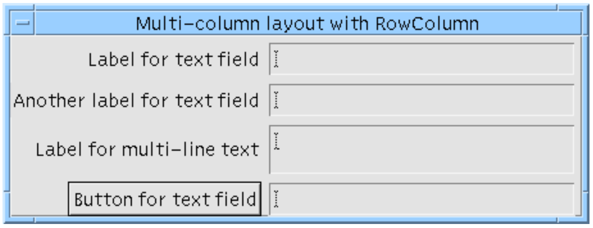

Multi–column layout with RowColumn
Label for text field
Another label for text field
Label for multi–line text
Button for text field

FIGURE 20-5 Multiple Column Layout with Form Widget

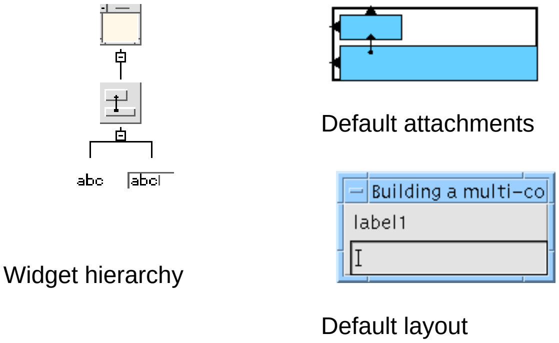
This layout cannot be achieved with a RowColumn because some of the rows are not all the same height. However, it can be achieved with a Form.

Column Layout Using the Form

The following section presents a systematic approach to creating complex column layouts. The first example is a two-column layout with a single row containing one Label and one TextField widget. Note that, unless otherwise noted, the value of all attachment offsets in this chapter is zero.

When you first create the Form and add its children, Motif arranges the children down the left side of the Form by attaching the left side of each widget to the left side of the Form. The top of the first widget is attached to the top of the Form and the top of each successive widget is attached to the bottom of the one above it. Therefore, if you create a Form containing a Label and TextField, you get the layout shown in Figure 20-6.

The Form used in this example has horizontal and vertical spacing set to 5 pixels.



Widget hierarchy

Default attachments

Default layout

FIGURE 20-6 Building a Multi-Column Layout

Figures 20-7 to 20-11 illustrate how to start shaping this arrangement into a two-column layout.

First, move the TextField to its approximate position as in Figure 20-7.

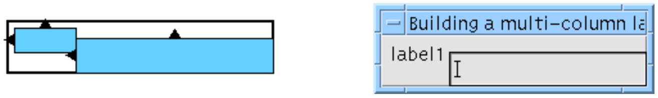


FIGURE 20-7 Positioning the TextField Widget

Next, align the top and bottom of the Label with the top and bottom of the TextField as shown in Figure 20-8.

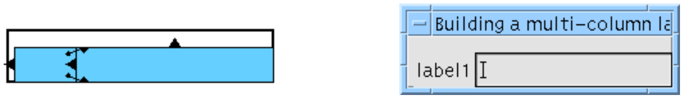


FIGURE 20-8 Aligning the Widgets

Attach the top and right side of the TextField to the top and right side of the Form, as shown in Figure 20-9.

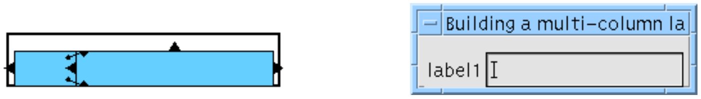


FIGURE 20-9 Attaching the TextField to the Form

Attach the right side of the Label to the left side of the TextField, as shown in Figure 20-10.

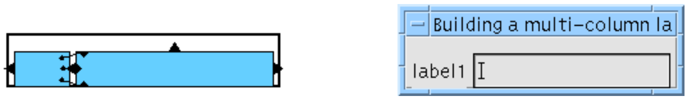


FIGURE 20-10 Attaching the Label to the TextField

Finally, set the position of the left side of the TextField at 25%, as shown in Figure 20-11.

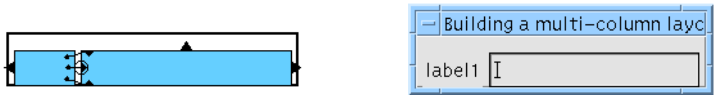


FIGURE 20-11 Setting the Position of the TextField

It may seem strange to fix the left side of the TextField by setting its position, while the right side of the Label is attached to the left side of the TextField. While it might seem more natural to fix the left side of the TextField by attaching it to the right side of the Label, this creates a circular attachment and the Form does not allow it. The 25% value used for the position is arbitrary at this stage. You cannot choose the final position until you know the widths of all the widgets, at least approximately.

Multiple Rows

It is easy to extend this procedure to multiple rows - just repeat the same steps. The only difference is that in the first row the extreme left and right positions (the left side of the Label and the right side of the TextField) are set by attaching them to the sides of the Form. For each subsequent row, however, these positions are set by aligning the widgets with the row above.

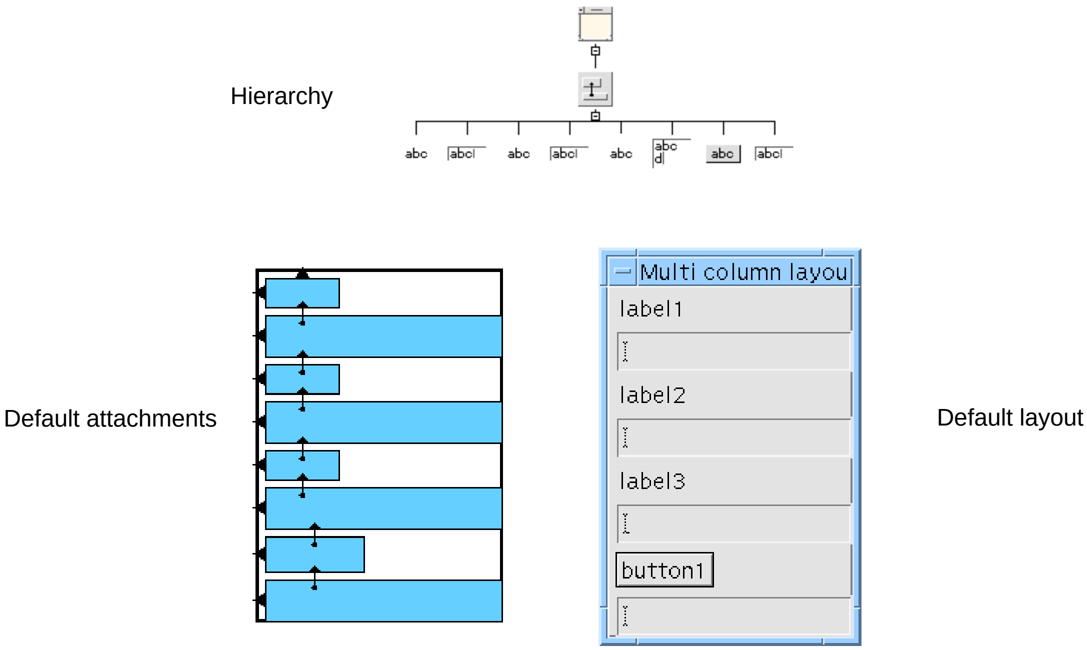


FIGURE 20-12 Multi-Column Layout – Initial State

Figure 20-12 shows the initial state of the dialog shown in Figure 20-5. In that dialog, the left column consist of three Labels and a PushButton while the right column contains three TextFields and a Text widget.

You can begin by setting the resources on the widgets that make up the dialog: the label text for the Labels and Pushbutton and the number of rows and edit mode of the multi-line Text widget.

It is not necessary to set these resources at this time; you can set them later if you prefer. However, setting them at this stage gives you an early impression of how the dialog will look and whether or not it is likely to work as you expect.

Set the resources to produce the layout shown in Figure 20-13.

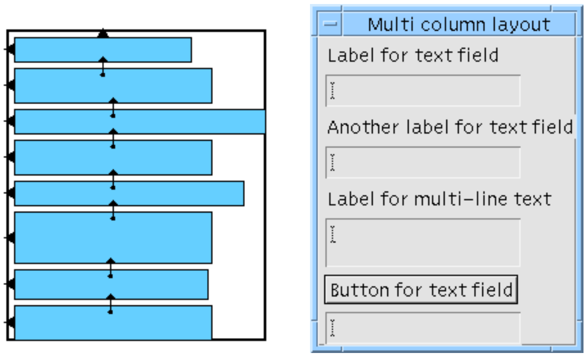


FIGURE 20-13 Multi-Column Layout with the Resources Set

Now you can apply the procedure illustrated in Figures 20-7 to 20-11 for each row of the column. The first step is to move the Text and TextField widgets to their approximate positions, as shown in Figure 20-14.

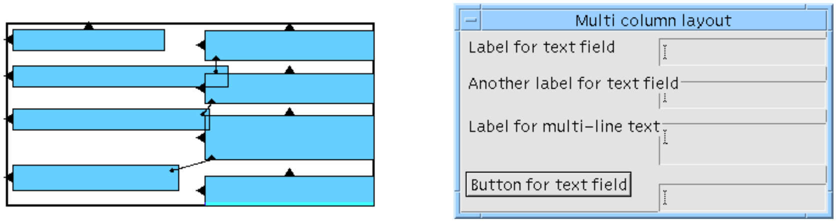


FIGURE 20-14 Approximate Positions

Next, align the top and bottom of the Labels and PushButton to the top and bottom of the corresponding Text or TextField widget as shown in Figure 20-15.

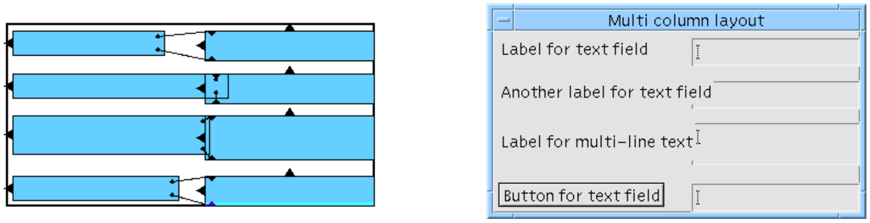


FIGURE 20-15 Aligning Labels and Text Widgets

As in Figure 20-9, you can now attach the TextField widget in the first row to the top and right sides of the Form using an offset of 5 pixels. Attach the top of each other TextField and Text widget to the bottom of the one above using an offset of 5 pixels.

Next, align the Text widgets with the top one. For the right sides, align the right side of each Text widget with the right side of the one above (or attach using an offset of 0 pixels). Alternatively, use “Group Align”, selecting the Text widgets from bottom to top. For the left sides, set positions to an arbitrary value such as 50%.

At this stage, align the left side of each Label and the PushButton with the left side of the widget above. Again, you can use “Group Align”, selecting the widgets from bottom to top. The layout should now resemble the one shown in Figure 20-16.

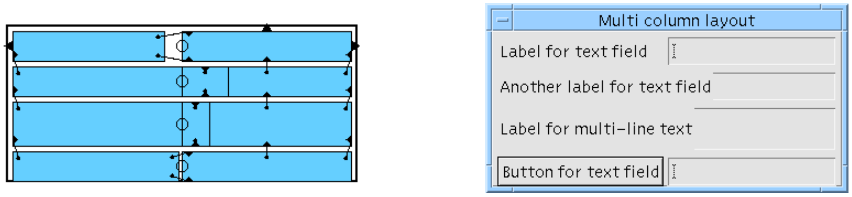


FIGURE 20-16 Aligning Into Columns

Attach the right side of each Label and the PushButton to the left side of the corresponding Text or TextField widget as shown in Figure 20-17.

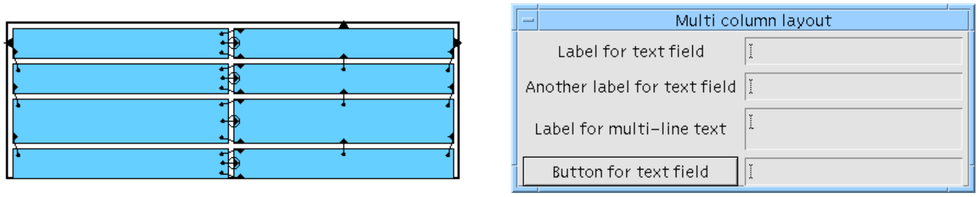
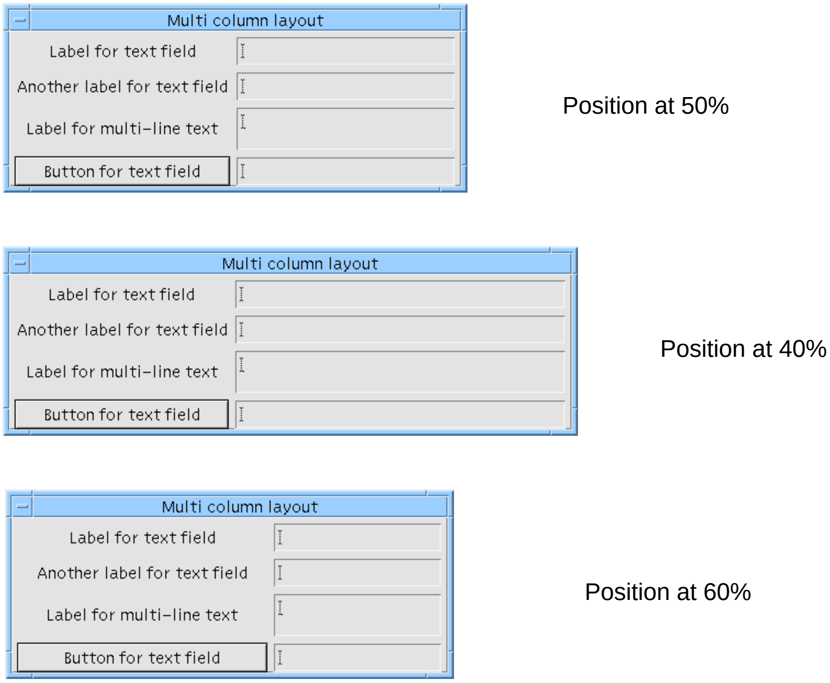


FIGURE 20-17 Labels Attached to Text and TextField Widgets

The final step is to adjust the position of the left side of the text widgets. This may require some trial and error. If the percentage is too high or too low, the Form is wider than necessary and wastes screen space. Some examples are shown in Figure 20-18.



Position at 50%

Position at 40%

Position at 60%

FIGURE 20-18 Setting the Position

Reset

Note that when you experiment with different position values, the Form may grow each time you change it (depending upon the value of the resize policy resource). This is because the Form is not too clever about constraints which change after the Form has been created. Reset the Form to see how it will really look in your application.

While the arrangement of attachments, alignments and positions used for the multiple column layout may seem complex, it is flexible and adaptable. In particular, it is relatively easy to add a new row to the dialog.

Adding a New Row

Adding a new row at the bottom of the dialog is easy; just add the new widgets to the design and set up the attachments as in the row above. Adding a new row in the middle of the dialog, however, is less straightforward.

Adding a Row in the Middle of the Dialog

The first step is to open a space for the new row. Because each row is only attached to the one above, and because the widget in the left column is attached at the top and bottom to the widget in the right column, you can move the whole bottom portion of the dialog down just by dragging the widget in the right column. This breaks all of its attachments to other widgets and you must remake them later. However, attachments from other widgets to the widget are unaffected. Figure 20-19 shows the effect of pulling down the Text widget.

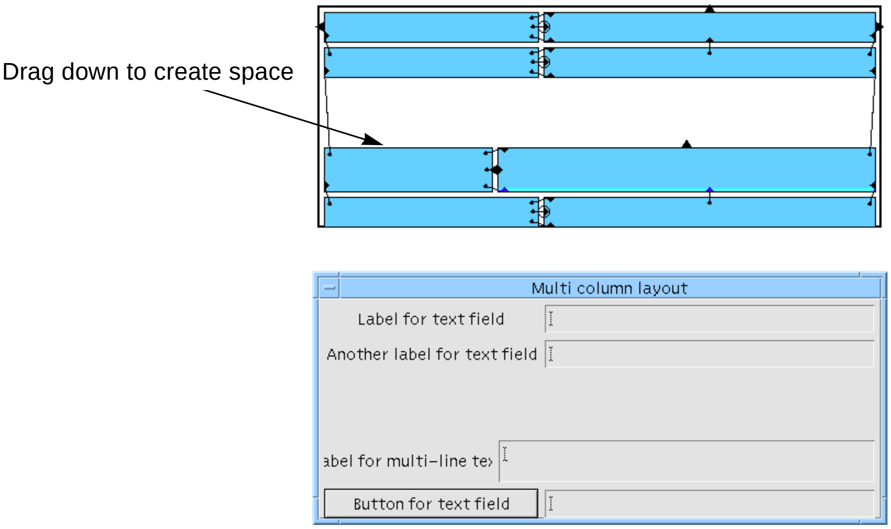
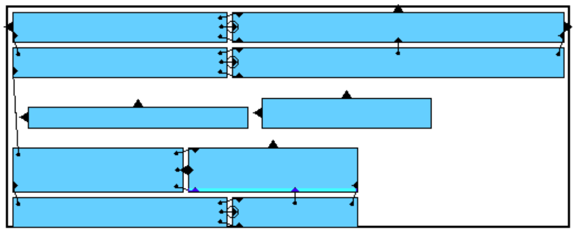


FIGURE 20-19 Making Space for a New Row

Now you can add the widgets for the new row, set their resources as required and move them to their approximate positions as shown in Figure 20-20.



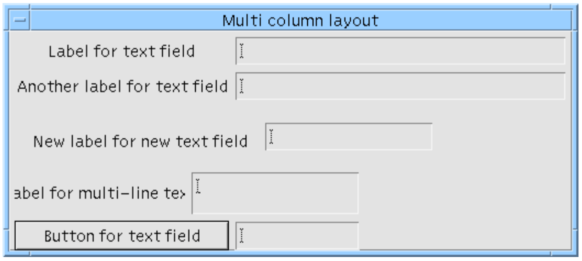


FIGURE 20-20 Adding the New Widgets

The next step is to align the widgets in the two columns as in Figure 20-21.

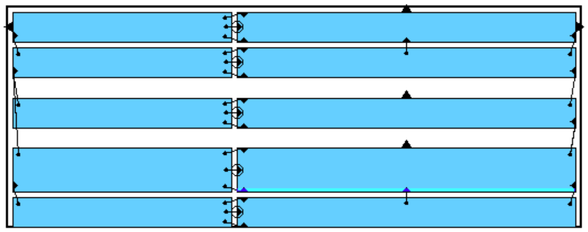


FIGURE 20-21 Alignments

Finally, attach each Text or TextField to the one above it and set the positions of the left sides of the Text or TextField widgets as in Figure 20-22.

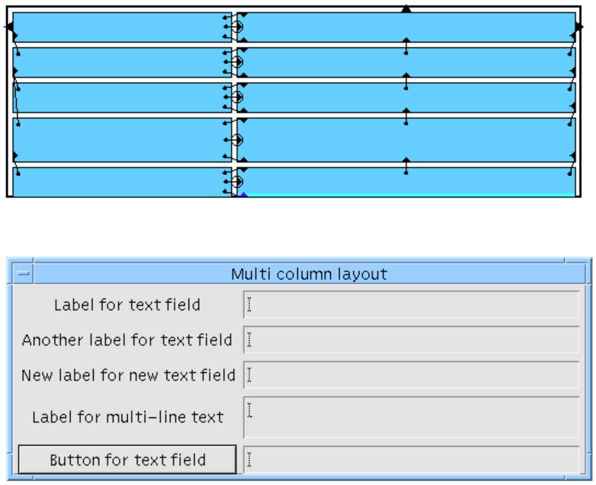

Multi column layout
Label for text field
Another label for text field
New label for new text field
Label for multi−line text
Button for text field

FIGURE 20-22 Attachments

As the new Label is a little too narrow, set the position of the left sides of the text widgets to 55% and reset the dialog to produce the result shown in Figure 20-23.

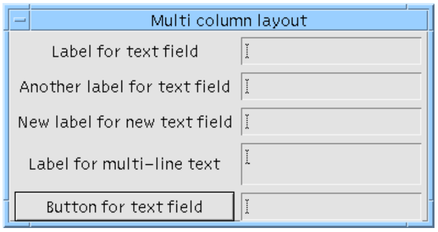

Multi column layout
Label for text field
Another label for text field
New label for new text field
Label for multi−line text
Button for text field

FIGURE 20-23 Position Set to 55%

Changing to Four Columns

Dialogs containing two columns, at least in part, are common. If there are so many items that a two-column layout becomes too tall, you can change it to a four-column layout.

For example, you can move the fourth and fifth rows of the example above into a new pair of columns, 3 and 4. The first step is to break the attachments, shown in Figure 20-24, that keep these rows aligned with the rows above.

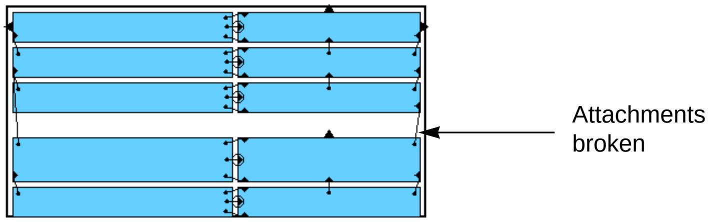


FIGURE 20-24 Breaking Attachments

Next you need to make room for two new columns on the right. This is done by setting positions on the first three rows which approximate the attachments they will have in the final Form. Figure 20-25 shows the result. When you change the positions, the Form becomes very wide and so you should reset the Form after changing the positions.

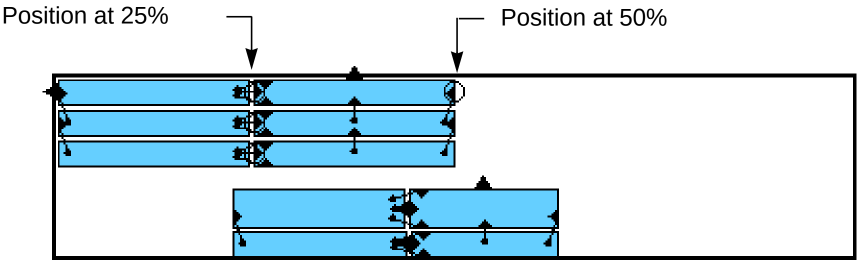


FIGURE 20-25 Reset Positions

Likewise, set approximate positions on the bottom two rows as shown in Figure 20-26. You must set the positions in the order shown to avoid temporarily putting the Form in a inconsistent state. If you do things in the wrong order, you get the message “Bailed out of edge synchronization”. While you can safely ignore this error message, you must dismiss the message box before continuing. Remember to reset the Form after changing the positions.

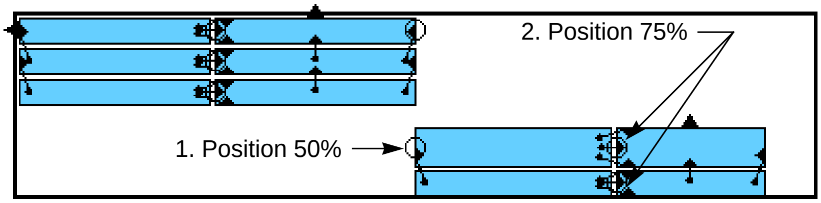


FIGURE 20-26 Position Right-Hand Columns

You can now move the right pair of columns up to their correct position by attaching the top Text widget to the top and right sides of the Form. This produces the result shown in Figure 20-27.

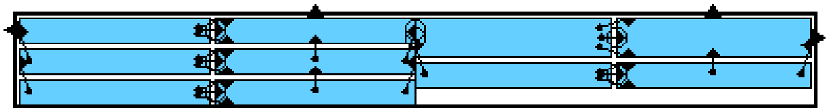


FIGURE 20-27 Attach to Top and Right of Form

The layout is almost finished. The right side of column 2 is currently positioned at 50%. Replace this position with an attachment to the left side of column 3, which is also positioned at 50%. Figure 20-28 shows the final layout.

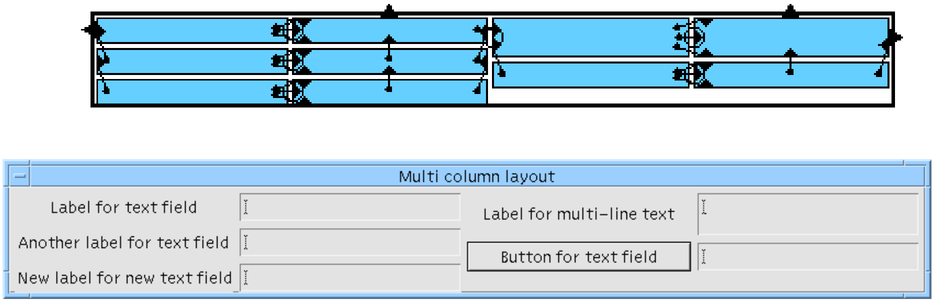

Multi column layout
Label for text field
Another label for text field
New label for new text field
Label for multi-line text
Button for text field

FIGURE 20-28 Final Layout

Edge Problems

A Form that is a child of a Shell draws a margin line round its inside edge. Any child widget of the Form that extends exactly to the edge of the Form occludes part of this margin line (Figure 20-29).

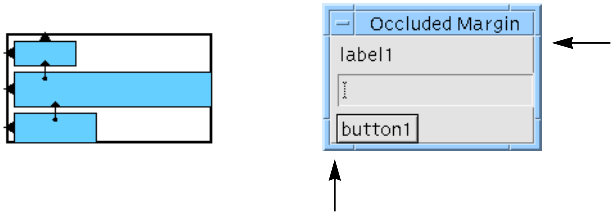


FIGURE 20-29 Occluded Margin

The simplest way to deal with this is to attach any widgets that overlap the margin to the sides of the Form with a small offset to reveal the margin. However, this can also produce undesirable resize behavior, as shown in Figure 20-30.

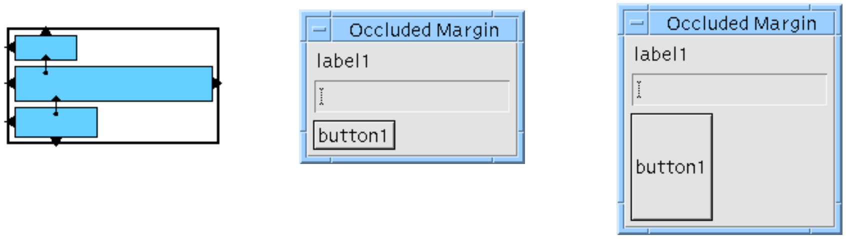


FIGURE 20-30 Extra Attachments and Resize Behavior

There are two other possible approaches to solving this problem, both of which involve introducing extra widgets into the design.

Invisible Widgets

The problem with the simple attachment in Figure 20-30 is that the button resizes when the Form does and it looks strange. An alternative is to introduce an extra, invisible widget. Although this also resizes with the Form, it does not look strange because it is invisible. A Separator gadget with the Type resource set to “No Line” is most effective. Figure 20-31 shows the widget hierarchy and attachments after adding an invisible widget to the previous example.

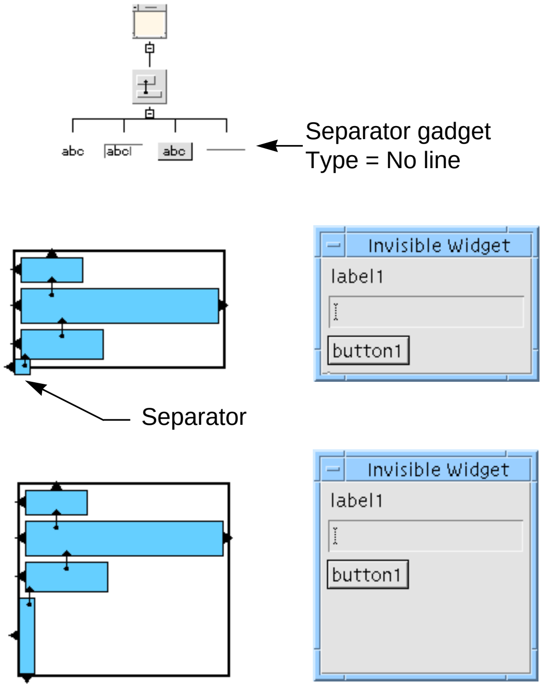


FIGURE 20-31 Invisible Widget

The bottom of the Separator is attached to the bottom of the Form by a small offset to prevent it from hiding the margin line. The top of the Separator is attached to the bottom of the PushButton with zero offset. The Separator now resizes vertically with the Form but the other components do not.

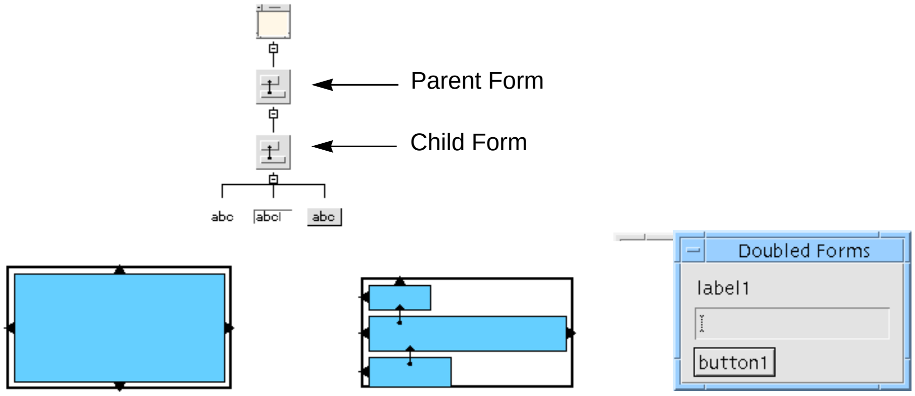
Note that the Form Layout Editor exaggerates the height and width of the Separator (and any other widgets under a certain size) to allow you to set its attachments using the mouse. Although this makes it look as if it is occluding the margin in the bottom left corner, in fact it is not.

You could also use a second Separator to keep the right side of the TextField widget inset from occluding the right edge of the Form (or even use the same Separator for both jobs). However, the horizontal resize behavior with the attachments shown in Figure 20-30 is appropriate for most purposes (the TextField widget resizes horizontally with the Form).

Doubled Forms

Another technique is to use a second Form as the invisible widget. This works because the Form only draws its margin line if it is the immediate child of a Shell. The intermediate Form, which is not a child of the Shell, does not draw a visible margin line and so its children can extend all the way to its edges without causing occlusion problems. Using an intermediate Form protects against margin occlusion on all four sides.

Figure 20-32 shows the widget hierarchy and appropriate attachments for this technique. The child Form is attached at all four sides to the parent Form with a small offset to make the margins visible.



Child Form attached to Parent Form

Widget attachments in Child Form

Final Result

FIGURE 20-32 Doubled Forms

As an alternative to the parent Form, you can use a BulletinBoard, setting the margin width and height to a relatively small value such as 5 pixels. Because the Shell assumes that its child is a BulletinBoard (or a BulletinBoard derivative such as a Form), you should not use a different kind of container widget (such as a Frame) in this position.

Form Resizing

What happens when the user resizes a Form depends on how the attachments and positions are set up. In general, you should try to design every dialog so that if the user makes it bigger, more of the important information is displayed. For example, if a dialog contains a scrolling list, the user should be able to make the visible portion of the list longer by making the window taller. On the other hand, there is no reason to make widgets such as Labels and buttons grow with the window, since this conveys no additional information.

In practice, any Form layout must probably compromise between resize behavior, robust response to size changes within the widget (such as font changes), ease of implementation and ease of maintenance. This section offers some basic guidelines for creating Forms with desirable resize behavior.

Widget Resizing

A widget in a Form grows wider when the Form does only if it has constraints on both right and left edges; it grows taller with the Form only if it has constraints on both top and bottom.

The most straightforward way to lay out a dialog is to work from the top left corner towards the bottom right corner of the Form, attaching the top and left of each widget to the bottom and right of a previous widget. If you do this, none of the widgets resize with the Form. To make the widgets in a Form resize sensibly, you must be a little more sophisticated.

Horizontal Resizing

You have already seen an example of horizontal resizing using the RowColumn widget in “Resize Behavior of RowColumns” on page 569. The rest of this chapter demonstrates some additional techniques.

Two-Widget Layouts

Layouts with only two widgets across the width of the Form are relatively simple. You only need to decide whether the extra width that results when the Form is resized is given to one widget or the other, or shared between the two.

Figure 20-33 shows one alternative. Widget 1 is attached to the Form at its left side but its right side is unconstrained; therefore, it finds its own natural width, wide enough to display the text label. Widget 2 is attached to Widget 1 on the left and to the Form on the right; therefore, its size varies to fill the part of the Form width that is not occupied by Widget 1. In other words, Widget 2 gets all the extra space.

When you reset the Form, it sets its own width to accommodate both widgets, producing the initial state shown in Figure 20-33.

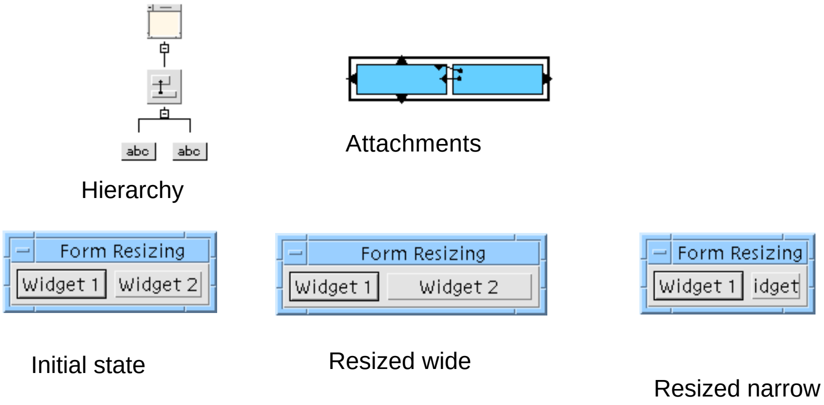
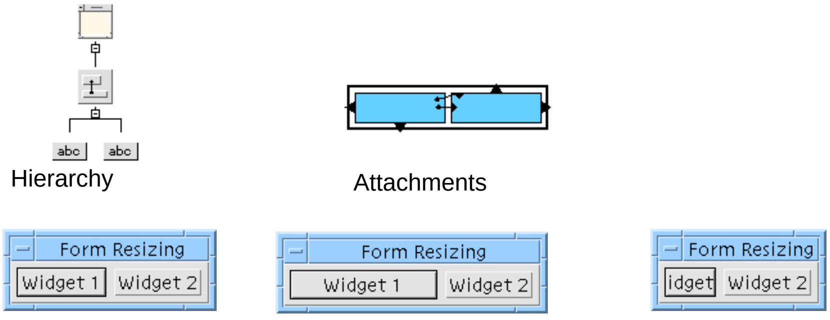


FIGURE 20-33 Two Widgets, Right Dominant

The attachments between the two widgets are reversed in Figure 20-34, i.e. the right side of Widget 1 is attached to the left side of Widget 2 instead of the other way around. The result is that Widget 2 stays the same size and Widget 1 gets all the extra space.



Hierarchy

Attachments

Initial state

Resized wide

Resized narrow

FIGURE 20-34 Two Widgets, Left Dominant

Avoiding Circularity When Reversing Attachments

You can reverse a right-to-left attachment between two widgets simply by adding a new left-to-right attachment in the same place. The Layout Editor detects this situation when you add the new attachment and removes the old attachment. However, in the example just given, you will see a circularity error message when you do this because there are two attachments to be changed. To get rid of the circular attachment, you must also swap the attachment that aligns the tops of the two widgets. Since both widgets have the same height, this does not affect the appearance of your layout.

Proportional Spacing

If you want to share the extra space equally between the two widgets, you must use proportional positioning. You can set a position on Widget 1 and attach Widget 2 to it or set a position on Widget 2 and attach Widget 1 to it, or set positions on both. Any of these methods works as long as you avoid circular attachments. Figure 20-35 shows the three possibilities.

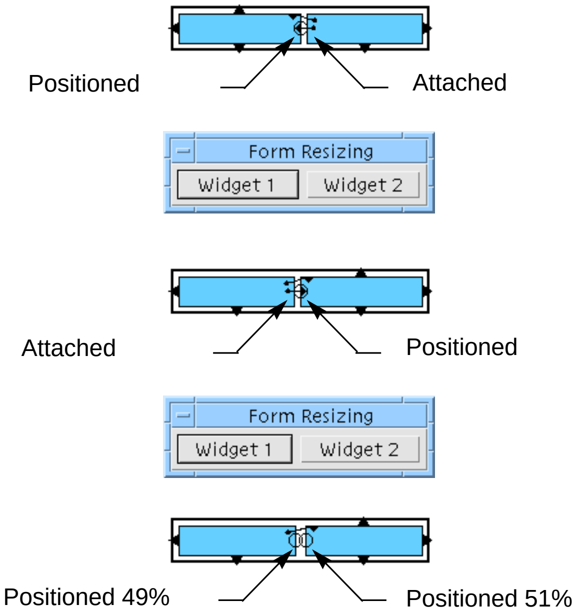


FIGURE 20-35 Two Widgets, Equal Shares

In Figure 20-35, the positions are set at about 50% and the two widgets share the width of the Form about equally. You can use different percentages to favor one widget or the other. In this example, the extra space is shared in proportion to the initial sizes of the widgets. The initial state and behavior of the form when resized narrow is the same in all cases.

Three-Widget Layouts

With three widgets, there are more possibilities. The extra space can be given to any one of the three, or shared among them. Figure 20-36 shows the simple cases, where all the extra width goes to one of the three widgets. Notice that in every case the widget with attachments or position settings at both ends is the one that resizes with the Form.

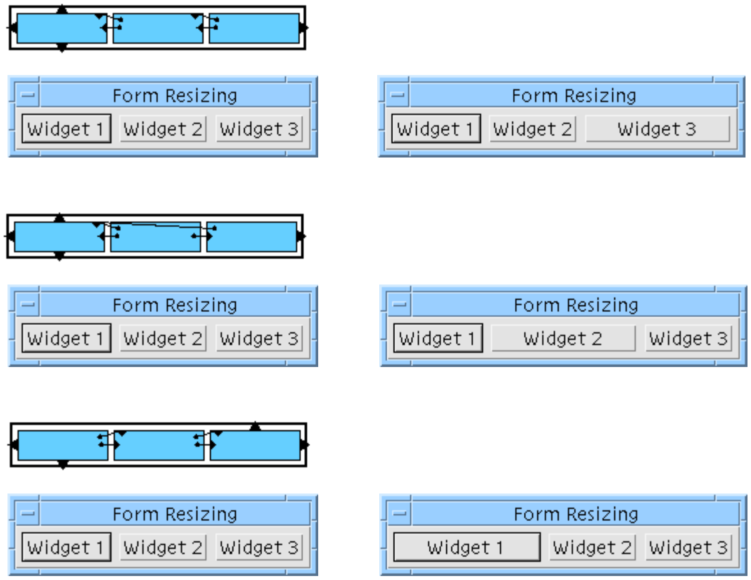


FIGURE 20-36 Three Widgets, One Dominant

To share the space among all three widgets, you can use simple proportional positioning, as shown in Figure 20-37.

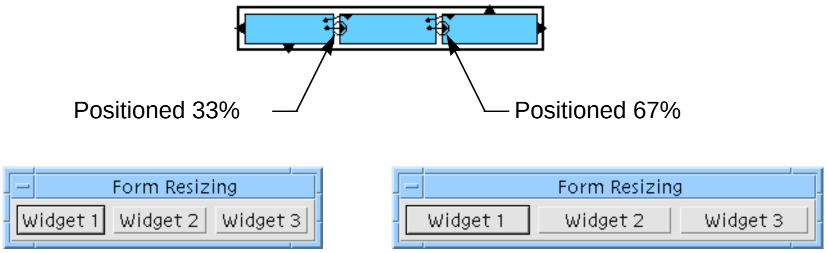


FIGURE 20-37 Three Widgets, Equal Shares

As with two widgets, you can use different percentages to favor one widget over another.

You can use combinations of attachments and positions to create layouts where one widget does not resize but the other two do. Figure 20-38 shows some examples.

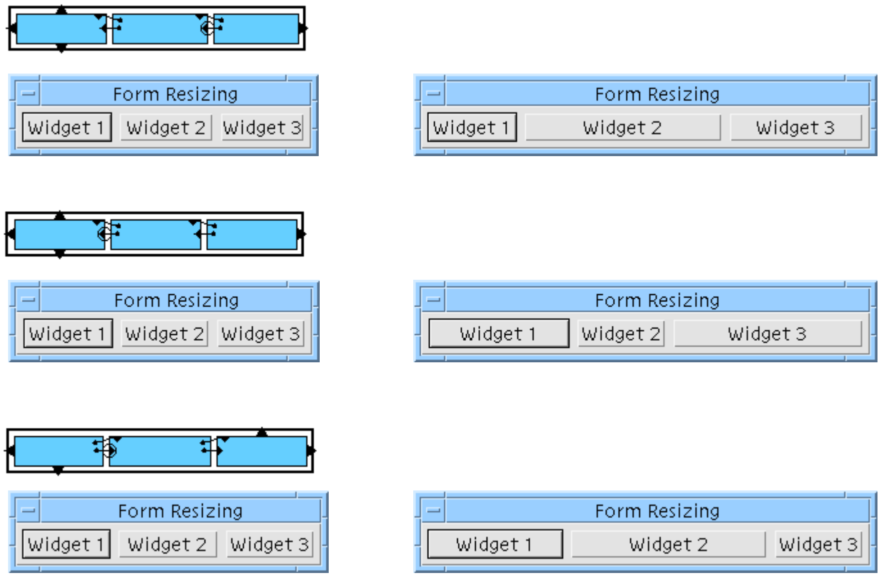
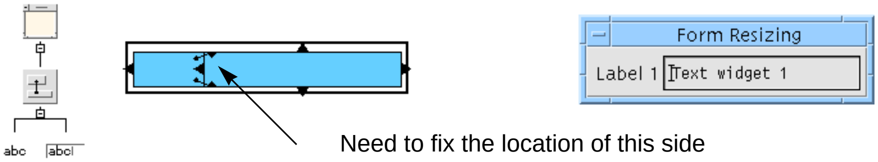


FIGURE 20-38 Combined Attachments and Positions

Widgets of Unequal Height

In all the examples so far, the widgets that share the width of the Form are all the same height. This gives you considerable freedom in arranging the attachments on the tops and bottoms of the widgets and lets you avoid circular attachments easily. If the widgets are of different heights, however, this is not so easy.

With a hierarchy like the one shown in Figure 20-39, you want a layout where the Text widget on the right makes maximum use of the available space, similar to the right-dominant layout in Figure 20-33. However, this requires attaching the widget on the right to the one on the left. You cannot do this in the example shown in Figure 20-39, because the Label widget on the left is already attached at the top and bottom to the Text widget to produce the correct vertical alignment.



Need to fix the location of this side

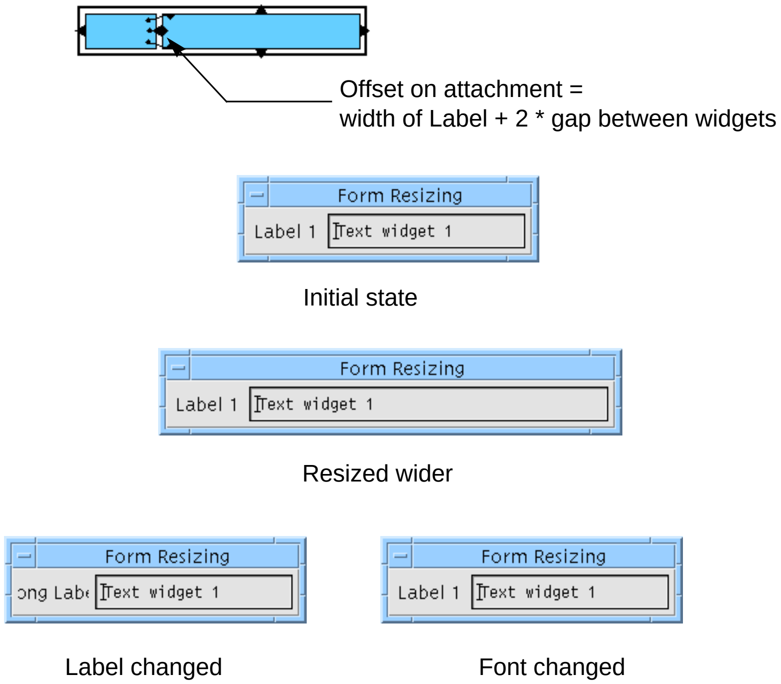
FIGURE 20-39 Two Widget Layout with Label and Text Widget

There are three ways to solve this dilemma:

1. Attach the left side of the Text widget to the left side of the Form using an offset large enough so that the Text widget does not obscure the Label. Attach the right side of the Label to the left side of the Text widget as shown in Figure 20-40.
2. Position the left side of the Text widget at a given percentage and attach the right side of the Label to it as shown in Figure 20-41.
3. If there is only a single row, the alignments at top and bottom of the Label can be replaced with attachments to the Form and the left side of the Text widget can then be attached to the right side of the Label as shown in Figure 20-42.

The figures show these three approaches and their behavior when the Form resizes, the label changes and a font changes. Variations on these approaches display similar behavior.

In Figure 20-40 the Text widget is attached to the Form at both ends. The virtue of this layout is that the Text widget gets all the extra space that results if the Form is resized wider. However, it is only satisfactory if the user is not likely to change the application's configuration extensively; it does not behave well if the label or the font changes.



Initial state

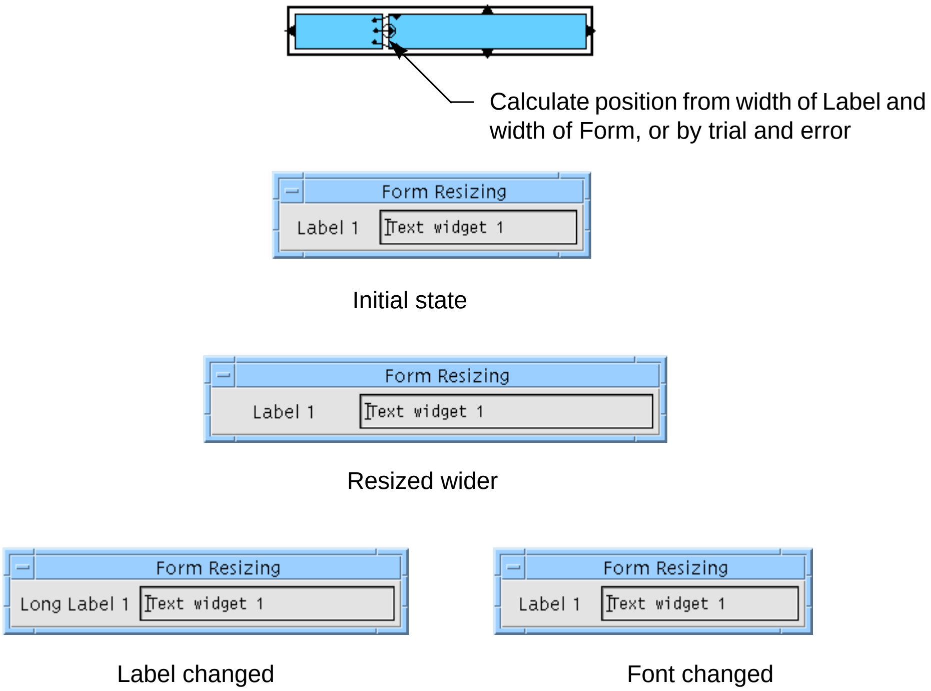
Resized wider

Label changed

Font changed

FIGURE 20-40 Text Widget Attached

In Figure 20-41, the left side of the Text widget is positioned. This is a more robust layout if the label or font changes as the label takes a share of any extra Form width. This is generally the most useful approach, especially for column layouts, as previously discussed.


Calculate position from width of Label and width of Form, or by trial and error

Initial state

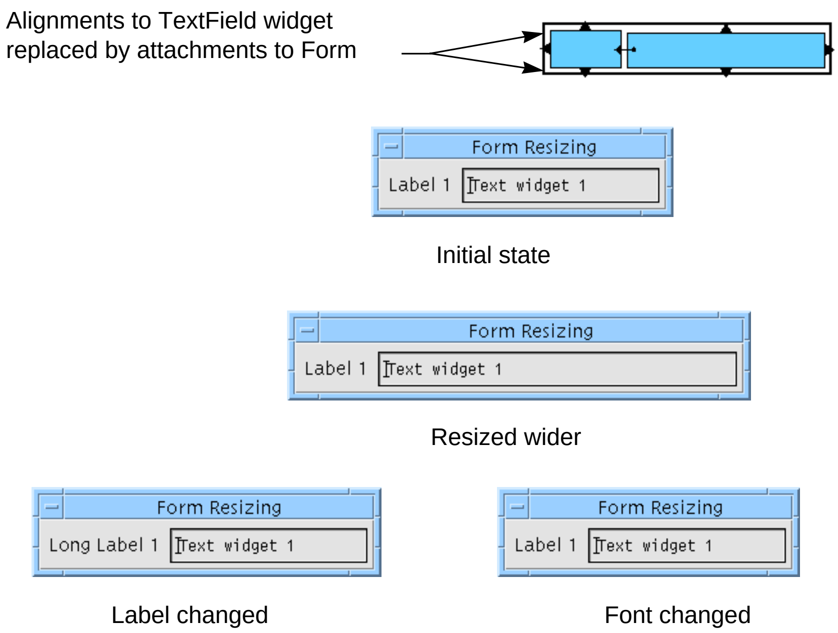
Resized wider

Label changed

Font changed

FIGURE 20-41 Text Widget Positioned

In Figure 20-42 the problem with circular attachments is removed by attaching the top and bottom of the label to the Form, instead of aligning them with the top and bottom of the Text widget. You can then attach the left side of the Text widget to the right side of the Label, producing a right-dominant layout similar to that used in Figure 20-33.



Alignments to TextField widget replaced by attachments to Form

Initial state

Resized wider

Label changed

Font changed

FIGURE 20-42 Removing Circularity

This solution exhibits the best behavior. However, you cannot use it for column layouts, since each row must be enclosed in a separate Form and there is then no way to align the columns vertically. It also requires an extra Form widget for each row, which can create significant overhead.

Vertical Resizing

The solutions for vertical resizing are essentially the same as for horizontal resizing. However, there are usually fewer problems with circular attachments. This is because a label positioned above an object can usually be aligned with the left edge of the object, while a label to the left of an object may need to be centered in the available space. Figure 20-43 illustrates this.

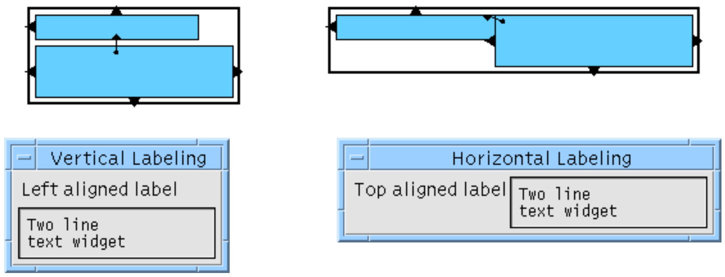


FIGURE 20-43 Vertical and Horizontal Clabel Placement

Figures 20-44 to 20-46 show examples of vertical layouts similar to the horizontal arrangements in Figure 20-36. In each case, any extra height is given to the multi-line Text widget.

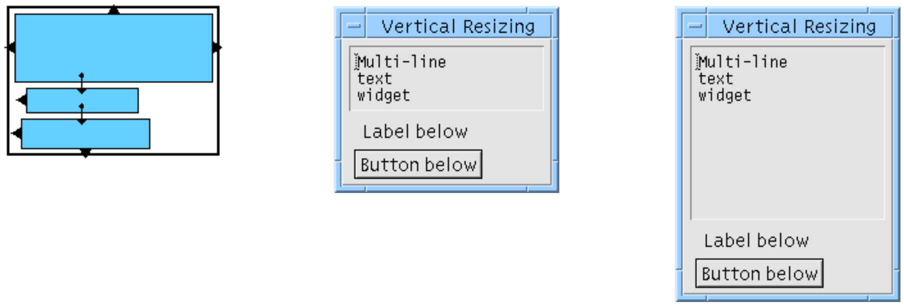


FIGURE 20-44 Resize Top Widget

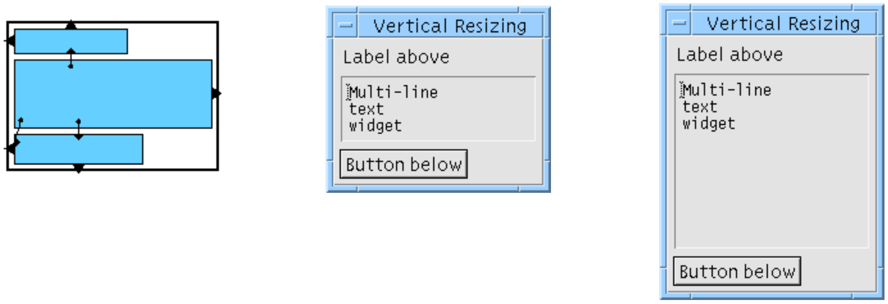


FIGURE 20-45 Resize Center Widget

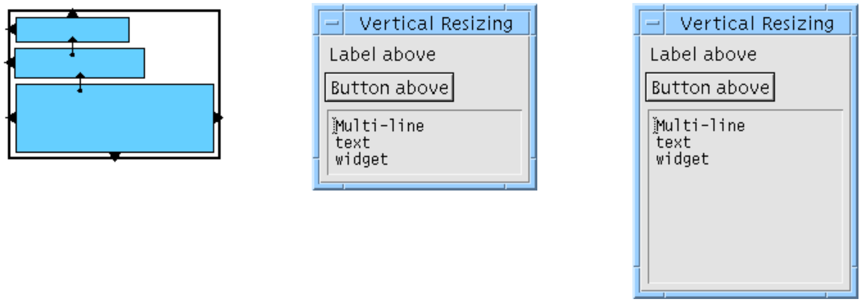


FIGURE 20-46 Resize Bottom Widget

Initial Size

The initial size of a dialog is determined by a process of negotiation between the Shell, the Form and the widgets within the Form. Normally the Form tries to find a layout that satisfies all the constraints on its children and lets each be at least as large as it wants to be, which is at least the “natural” size. The Form then sizes itself to contain this layout and the Shell sizes itself to contain the Form.

Most widgets have a sensible natural size. Labels, for example, have a natural size determined by the text content of the label and the font; Text widgets normally size themselves according to the number of rows and columns specified by their resources.

If your dialog only contains widgets that have an acceptable natural size, you can let the Form work out the initial size for you. Problems only arise when the dialog contains widgets that do not have an acceptable natural size. You must then fix their size with constraints, which may make the initial size of the dialog seem too small.

If you have this problem, set the initial size of the dialog by setting the width and height resources of the top level Form or BulletinBoard. You cannot set the initial size using the width and height resources of the Shell, although you can set a minimum size using the appropriate Shell resources.

CHAPTER 21

Hypertext Help

Introduction

Sun WorkShop Visual provides extensive on-line help which can be accessed from many different points in the application. This chapter describes the support that Sun WorkShop Visual provides for building help into your application.

The Help Model

The model used to support help in your application is very simple. You can either use the help callback, available with all Motif widgets, or an activation callback on a help button. Therefore, to provide context-sensitive help, all you need is a generic callback that takes as its client data the help (or a path to the help) to be displayed when the callback is invoked. In Sun WorkShop Visual this help specification is defined as a document path and a marker that denotes some reference in that document. We supply a callback that can be used with one of the help system interfaces provided with Sun WorkShop Visual. See “Help Viewers” on page 598 for more information on which systems are provided to display your help.

Obviously you are free to re-implement the callback that you use in your application to make it interface with the help system of your choice. Likewise, you can achieve the same effect simply by using the ordinary callback mechanisms, or you can decide not to give your users any on-line help at all.

Internally, Sun WorkShop Visual uses two systems when displaying help:

- Sun WorkShop Visual Help. This is the help viewer with built-in navigation commands and links to related help topics.
- Netscape. This is used as a HTML browser.

In order to try out your help within Sun WorkShop Visual you will have to use one of these.

Motif's Help Callback

By default in Motif the Help callback is invoked by the Help action which is specified for every widget. This action is bound to the <*osfHelp*> key using a default translation in every Motif class.

Help Viewers

Sun WorkShop Visual provides interfaces to the following viewers:

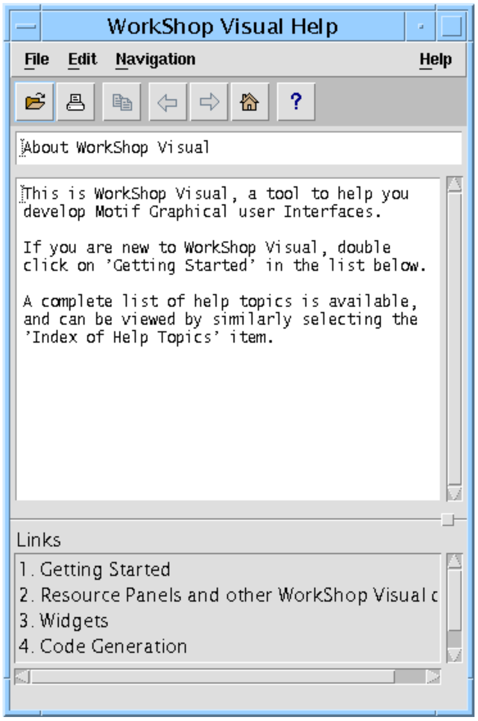
- Sun WorkShop Visual Help
- Netscape
- FrameMaker

You can use these to display help in your application. They are explained in more detail in the following sections.

You set up help for your application in the same way, regardless of which viewer you intend to use. You then link your application with the appropriate supplied library, as described in "Help Implementation" on page 613.

Sun WorkShop Visual Help

Sun WorkShop Visual Help is Sun WorkShop Visual's own help viewer and is shown in Figure 21-1.



About WorkShop Visual

This is WorkShop Visual, a tool to help you develop Motif Graphical user Interfaces.

If you are new to WorkShop Visual, double click on 'Getting Started' in the list below.

A complete list of help topics is available, and can be viewed by similarly selecting the 'Index of Help Topics' item.

1. Getting Started
2. Resource Panels and other WorkShop Visual c
3. Widgets
4. Code Generation

FIGURE 21-1 Sun WorkShop Visual Help Viewer

The Sun WorkShop Visual Help help viewer contains a menubar with “File”, “Edit”, “Navigation” and “Help” menus containing file, editing and navigation commands respectively. The “Help” menu contains version information and help on using Sun WorkShop Visual Help. A toolbar is also provided with buttons for quick access to most of the menu items. As you pass the mouse pointer over the toolbar buttons, information about the function of the button is displayed in the status line at the bottom of the viewer window.

There are two text areas in the help viewer. The topmost text area displays help about the currently selected topic. The bottom area contains links to other related topics. Double clicking over one of the links displays help on that topic.

The files used by Sun WorkShop Visual Help are in HTML (HyperText Markup Language). This is a public domain format which uses only printable characters and can, therefore, be created in any text editor. It is also a standard used by many applications.

Sun WorkShop Visual Help does not claim to be a full HTML interpreter; it recognizes a subset of HTML and interprets them in a simple way. “HTML Tags” on page 601 describes the HTML keywords recognized by Sun WorkShop Visual Help and the way they are interpreted.

See Appendix E, “Further Reading” for suggested books on HTML.

Netscape

Netscape is a browser which takes HTML (HyperText Markup Language) files and displays them complete with any hypertext links or special formatting you have specified.

Netscape also provides file operations and navigation commands allowing the user to browse around the help files you provide.

FrameMaker

FrameMaker is a desktop publishing package which uses its own internal format. FrameMaker also provides a means of viewing and browsing around read-only documents. For information on using FrameMaker as your help viewer, see “Using FrameMaker” on page 604.

Using HTML in Help Documents

Both Sun WorkShop Visual Help and Netscape read HTML files. You will need to write the files so that you have specified HTML *anchors* at points which will be the source and destination of a link. You should remember the names of these anchors as they are used to specify the help action for individual widgets.

These anchor points are the “markers” in Sun WorkShop Visual which are specified in the Help callback. Setting up Help callback markers is explained in “Setting up Help in Sun WorkShop Visual” on page 605.

To use either Sun WorkShop Visual Help or Netscape as the help system in your application you will need to do the following:

1. **Create the help files in HTML format**

   You do not have to create a separate file for each topic as, in HTML, you can have links within the same document. In order to maintain the help text, however, you may wish to have separate files. Because Sun WorkShop Visual Help uses a subset of HTML, you will need to refer to “HTML Tags” on page 601 for details on which HTML keywords are recognized by Sun WorkShop Visual Help.

2. **Specify the Help callback for individual widgets or for whole modules**

   See “Setting up Help in Sun WorkShop Visual” on page 605 for information on adding help callbacks to widgets in your design. Note that you will need to follow the instructions in “Linking Help Into Your Application” on page 611 so that the help callback is defined in your application.

3. **Link the generated code with the Sun WorkShop Visual Help or Netscape libraries provided**

   See “Help Implementation” on page 613.

HTML Tags

This section gives a brief description of the major HTML keywords, known as *tags*. The tags listed here are all those interpreted by Sun WorkShop Visual Help. The description given offers an indication of the way the tags are interpreted by most full HTML interpreters (such as Netscape) and the way they are interpreted by Sun WorkShop Visual Help, if different.

In HTML tags are enclosed within angle brackets (< and >). You must use these brackets in your text files. See “Example HTML Document” on page 604 for an illustration of the way you should use HTML tags in your documents.

The tags have been divided into categories according to their usage. Note that the tags are listed here in uppercase. HTML, however, is not a case sensitive language. You can, therefore, use either case.

Starting and Ending

<HTML> All HTML documents start with this tag.

</HTML> HTML documents end with this.

Anchors and Links

<A NAME=”*anchorname***”>***text***</A>** This is an anchor which will be used as the destination of a link. *anchorname* is your internal name for the anchor. *text* is the text you want people to click on to go to the link destination.

<A NAME=”*#anchorname***”>***text***</A>** This is the source of a link to another part of the same document. *anchorname* is the internal name given to the anchor.

You can have links to other files using relative or absolute pathnames. An external file can even reside on another server. For external links *anchorname* is the filename and you should omit the ‘#’ sign at the beginning.

Paragraph

<P> This tag is used to indicate the start of a paragraph. Paragraphs are separated by a blank line.

</P> This tag is used to indicate the end of a paragraph.

Break

<BR> This tag indicates a hard line break. If there is no line break, the text will flow to fit the width of the browser window.

Lists

Sun WorkShop Visual Help does not handle the different types of list in the way most HTML interpreters do. Lists are interpreted for compatibility so that existing HTML documents can be used.

<OL> Indicates that the following lines are an ordered list. Most HTML implementations would put a number in front of the list items but Sun WorkShop Visual Help simply prints a ‘-’ (dash).

</OL> Indicates the end of an ordered list.

<UL> Indicates that the following lines are an unordered list. Most HTML implementations would put a bullet mark in front of the list items but Sun WorkShop Visual Help simply prints a ‘-’ (dash).

</UL> Indicates the end of an unordered list.

<DL> Indicates that the following lines are a definition list. Most HTML implementations would expect a “term” and a “definition” as in a glossary - Sun WorkShop Visual Help simply prints a ‘-’ (dash) in front of each item in the list.

</DL> Indicates the end of a definition list.

<LI> Indicates a list item. A ‘-’ (dash) is printed at the beginning of the line.

<DT> Indicates a “term” in a definition list. Sun WorkShop Visual Help treats these as plain list items.

<DD> Indicates a “definition” in a definition list. Sun WorkShop Visual Help treats these as indented list items with no ‘-’ (dash).

Character Formats

<EM> Indicates the following text should be emphasized. Some HTML interpreters use italic and some use bold. Sun WorkShop Visual Help prints a ‘*’ (asterisk) before and after the text.

</EM> The end of the text to be emphasized.

<B> Indicates the following text should be printed in bold. Sun WorkShop Visual Help prints a ‘*’ (asterisk) before and after the text.

</B> The end of the text to be printed in bold.

<I> Indicates the following text should be printed in italic. Sun WorkShop Visual Help prints a ‘*’ (asterisk) before and after the text.

</I> The end of the text to be printed in italic.

Headings

HTML defines a range of headings from “H1” to “H6” which, according to most HTML interpreters, start large and bold and gradually lessen in size and weight. Sun WorkShop Visual Help simply prints two blank lines before a heading. The heading tags indicate the beginning and end of the text of the heading:

<H1>First Level Heading**</H1>**

<H2>Second Level Heading**</H2>**

<H3>Third Level Heading**</H3>**

<H4>Fourth Level Heading**</H4>**

<H5>Fifth Level Heading**</H5>**

<H6>Sixth Level Heading**</H6>**

Preformatted Text

<PRE> Most HTML interpreters format the text in a HTML document ignoring any extra spaces, tabs or line returns. The “preformatted” tag causes the interpreter to print the text as you typed it and uses a monospaced font, such as Courier. Sun WorkShop Visual Help only prevents the output function from stripping white space.

</PRE> End of preformatted text.

Example HTML Document

Below is an example of a help document written for Sun WorkShop Visual Help in HTML:

```
<html>
<head>
About Sun WorkShop Visual
</head>
<h1>About Sun WorkShop Visual </h1>
<p>This is Sun WorkShop Visual, a tool to help you develop Motif
Graphical user Interfaces.
<p>
If you are new to Sun WorkShop Visual, select ‘Getting Started’ in the
 list below, and then press the Follow link button.

<p>
A complete list of help topics is available, and can be viewed by
similarly selecting the ‘Index of Help Topics’ item.

See also:
<ul>
  <li><a href=”get_started”>Getting Started</a><br>
 <li><a href=”dialogs”>Resource Panels and other >Sun WorkShop Visual
dialogs</a><br>
  <li><a href=”widget_list”>Widgets</a><br>
  <li><a href=”code_gen”>Code Generation</a><br>
  <li><a href=”index”>Index of Help Topics</a><br>
</ul>
</html>
```

This file is shown displayed in Sun WorkShop Visual Help in Figure 21-1.

Using FrameMaker

If you plan to use FrameMaker to build your help system, you must know how the FrameMaker hypertext system works. Complete documentation is provided in your FrameMaker manuals and a brief summary is given below.

FrameMaker lets you mark places in the text as either source or destination markers. Destination markers are places you can jump to in a help document. Source markers are places in a help document or your application that the user can select which cause an action, usually a jump to a destination marker. Source markers in a help document require a visual clue to tell the user he can click on them. The best way to do this is to specify a character format such as italics or underline to denote a source link.

When the user clicks in a document at a place where the special character format is in effect, FrameMaker checks for a source marker in that area. If it finds one, it highlights the whole graphical area or section of text and executes the action specified by the marker.

You can insert these special markers into your FrameMaker document by using the “Marker” command from the Special menu.

1. **Select “Hypertext” from the Marker Type list.**
2. **To specify a destination marker, type:** `newlink <tag_name>` **in the Marker Text field.**
3. **To specify a source marker, type:** `gotolink <tag_name>`

   A tag can be specified either as *<tag_name>* for a tag in the current document, or as *<document_name>:<tag_name>* for a tag in a different document.
4. **Hypertext markers are only effective in a locked document. To toggle the lock of a document on or off, type:** *<escape> <F> <l> <k>*
5. **To exit from a locked document, type:** *<escape> <f> <c>*

   This command gives you the option of saving the document in its locked state.

Setting up Help in Sun WorkShop Visual

The term *tag* means the combination of a document and a marker. Help Tags are specified from the “Code generation” page of the Core resource panel. There are two Help tags that can be specified: one for the Help callback, which can be specified for any Motif widget, and one for the Activate callback, which can only be specified for PushButton and CascadeButton widgets.

Note – The Activate callback for a CascadeButton is only called if there is no pulldown menu associated with it.

To specify a tag, type the name of the document and the name of the marker, and press “Apply”. You should now be able to display that document by invoking the callback in the dynamic display, which is done by clicking on the button for an Activate callback, or pressing *<F1>* for a Help callback.

When you specify tag information in the Core resource panel, Sun WorkShop Visual automatically adds the callback function *XDhelp_link()* to the appropriate callback list for the widget. You do not have to specify a callback in the Callbacks dialog for the widget.

Inherited Documents

You may be able to specify some of your help information by typing the marker only, without having to retype the name of the help file. If you specify a marker but no document name for the Help callback, Sun WorkShop Visual uses the Help callback document of the closest parent widget. If none of the widget’s ancestors has a Help callback document, Sun WorkShop Visual uses the default document specified in the Module Help defaults panel, as described in the *Module Defaults* section below.

In the case of the Activate callback, if no document is explicitly set the document for the Help callback is used. If this is not set, the same Help document inheritance, described above, takes place.

Help on Windows

If you are running Sun WorkShop Visual in Windows mode, you can specify Activate Help callbacks which are then mapped through to the Windows code.

When the generated application is run on Windows, these callbacks invoke the default web browser on the system using the specified URL and a pre-defined help path. The help path is specified in the Help Defaults dialog which is displayed by selecting “Help Defaults” from the Module menu. The default path should be set to the root of the HTML included with the application.

To allow the HTML files to be opened on multiple platforms, you need to be able to modify the path to the root of the HTML according to the platform. For example, the path “`c:\program files\myapp\myhtml`” will not work with the Motif XP version of the help system. In order to help you with this, Sun WorkShop Visual generates the following structure into the stubs and the main code file.

In the main code file the following is generated:

```
#ifndef DUAL_PLATFORM
char * _xd_help_path = "c:\\program files\\myapp\\myhtml\\"
#else
extern char * _xd_help_path;
#endif
```

The stubs file contains a similar structure:

```
#ifndef DUAL_PLATFORM
extern char * _xd_help_path;
#else
char * _xd_help_path = "c:\\program files\\myapp\\myhtml\\"
#endif
```

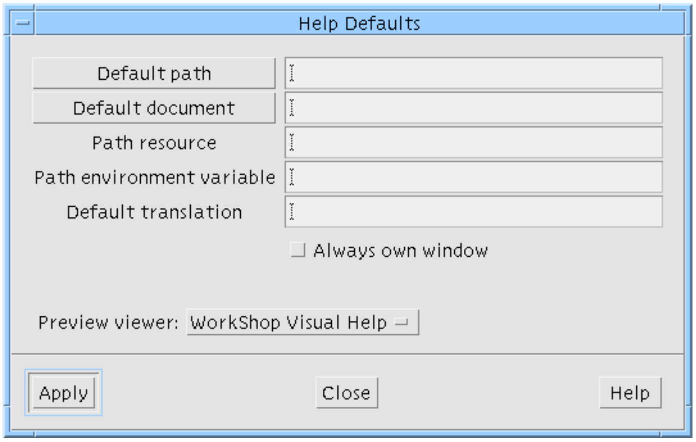
This structure can be modified and extended. Modifications are retained when code is regenerated. This allows you to specify the help path for any system you intend to use for the help, as shown in the following example:

```
#ifndef DUAL_PLATFORM
extern char * _xd_help_path;
#else
#ifdef WIN32
char * _xd_help_path = "c:\\program files\\myapp\\myhtml\\"
#else
char * _xd_help_path = "/opt/myapp/HTML"
#endif
```

You then need to specify `DUAL_PLATFORM` in the project settings for your application. If you do not wish to provide absolute paths, you can use this mechanism to pick up environment variables or settings from the registry.

Module Defaults

There are numerous help defaults that can be specified on a per module basis. To specify these defaults, pull down the Module Menu and select “Help defaults”. The resulting dialog is shown in Figure 21-2.



□ Always own window

Preview viewer: WorkShop Visual Help

FIGURE 21-2 Help Defaults Dialog

The Help Defaults Dialog lets you specify defaults for use both in building the help system in Sun WorkShop Visual and in the finished application.

TABLE 21-1

Default document:	This field specifies the name of the default document for use if no other document is specified on an individual widget.
Default path:	All help tag document names are assumed to be relative paths unless they begin with “/”. This field specifies a default path to be added to the beginning of any relative path document names. The Default path is also used to find documents when you test your help in Sun WorkShop Visual.
Path resource:	In the application, you can override the Default path setting by setting an application resource. This field specifies the name of the application resource.

TABLE 21-1 *(Continued)*

Path environment variable:	You can also override the resource setting by setting an environment variable. This field specifies the name of the environment variable.
Default translation:	This field specifies an event to be added as a translation to every widget that has a marker specified for the Help callback. The translation calls the *Help()* action. This lets you designate a key combination as an application help key.
Preview Viewer	This option menu lets you specify which help viewer Sun WorkShop Visual should use when you ask to preview your help document(s). There are three options: Sun WorkShop Visual Help, Netscape and FrameMaker.
Always own window:	This toggle makes the FrameMaker integration callback display the document in its own window. If this toggle is off, an existing window is used to display the new page. You can designate the use of a new window for individual pages using the “Own window” toggle in the Help documents and markers dialog.

Finding Help Documents and Markers

The “Activate callback” and “Help callback” buttons in the “Code generation” page of the Core resource panel can be used to display a dialog showing all the documents and markers that are currently referenced by your design.

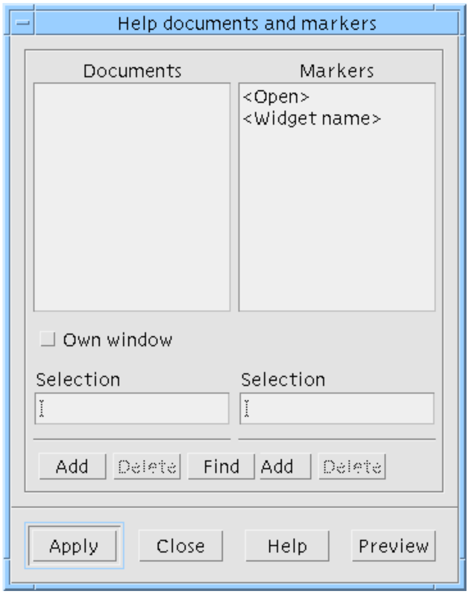


FIGURE 21-3 Help Documents and Markers Dialog

TABLE 21-2

Add:	You can add a document or marker by typing the name in the appropriate selection field and pressing “Add”.
Delete:	You can delete a document or marker by selecting its name from the list and pressing “Delete”. You cannot delete documents or markers that are still referenced by a widget in the design.

TABLE 21-2 *(Continued)*

Find:	This button pops up a file selection dialog to help you navigate in the file system. You can also display this dialog by selecting the “Default path” or “Default document” buttons in the Help defaults dialog.
Preview:	This button makes Sun WorkShop Visual try to connect to the selected help viewer and display the named document at the named marker. Select which help viewer you wish to use in the Module Help Defaults dialog - see “Module Defaults” on page 608.
Own window:	Each document has an “Own window” flag associated with it. This flag forces the system to display this document in its own window, not to share a common window with other documents. This toggle lets you designate the state of the flag for the selected document.
<*Open*>:	A special default marker that causes the document to be opened at the first page.
<*Widget name*>:	A special default marker that uses the widget name as a marker to jump to in the document.

Linking Help Into Your Application

To use Sun WorkShop Visual’s default help callback function, you must link in the supplied object files which can be found in Sun WorkShop Visual’s installation directory (referred to below as $VISUROOT).

There are three directories, one for each help viewer. For Sun WorkShop Visual Help and FrameMaker you will also need to link with additional files.

Sun WorkShop Visual Help

To use Sun WorkShop Visual Help you will need to link the file `helplink.o` in with your application. The source of this object file, `helplink.c,` is found in:

```
$VISUROOT/src/libhelplink/helplink
```

There is also a `Makefile` with instructions for building `helplink.o` and a file named README with information on linking in Sun WorkShop Visual Help.

In addition you will need to link in the library in:

```
$VISUROOT/src/help/client
```

There is a `Makefile` in that directory with instructions for building the library.

Netscape

To use Netscape you will need to link the file `helplink.o` in with your application. The source of this object file, `helplink.c,` is found in:

```
$VISUROOT/src/libhelplink/nshelplink
```

There is also a `Makefile` with instructions for building `helplink.o` and a file named `README` with information on linking in Netscape.

FrameMaker

To use FrameMaker you will need to link the file `helplink.o` in with your application. The source of this object file, `helplink.c,` is found in:

```
$VISUROOT/src/libhelplink/fmhelplink
```

There is also a `Makefile` with instructions for building `helplink.o` and a file named `README` with information on linking in FrameMaker.

In addition you will also need to build the files in

```
$VISUROOT/src/libhelplink/fmhelplink/libframe/fmclient
```

There is a `Makefile` in that directory with instructions for building the library.

Other

You can also supply your own Help callback function. This function should also be called *XDhelp_link()* and should follow the same form as Sun WorkShop Visual's function. See the *Help Implementation* section below for suggestions on how to customize the Help callback function.

Help Implementation

The preceding descriptions show how the help system works within Sun WorkShop Visual. If you use the default Help callback provided with Sun WorkShop Visual, the help in your application will work in the same way. However, you may want to customize your implementation of the Help callback. The following sections describe how to do so using the FrameMaker integration as a guide.

The Sun WorkShop Visual directory *libhelplink/fmhelplink* contains all the sources for the FrameMaker integration callback *XDhelp_link*. The subdirectory *libframe* contains various source files, adapted from the FrameMaker release. The original files are in *$FMHOME/source/openmaker* if you want to check them out.

XDhelp_link() receives a *client_data* parameter that is a pointer to an *_XDHelpPair_t* structure:

```
typedef struct _XDHelpPair_s {
_XDHelpDoc_p doc;
char ** tag;
Bool  open_doc;
} _XDHelpPair_t, *_XDHelpPair_p;
```

This contains a pointer to an *_XDHelpDoc_t* structure, a pointer to a marker (*tag*), and an *open_doc* flag. If tag is *NULL* the developer has specified one of the default markers: <*Widget name*> if *open_doc* is *FALSE*, <*Open*> if *open_doc* is *TRUE*. The document structure is:

```
typedefstruct_XDHelpDoc_s{
char * doc;
char ** path;
int  handle;
Bool new_window;
}_XDHelpDoc_t,*_XDHelpDoc_p;
```

The *doc* field is the name of the document. *path* is a pointer to the default path if one exists. This path is calculated as described above, taking account of the setting of the help path application resource and environment variable. *handle* is the document handle returned from FrameMaker, and *new_window* specifies whether the document is to have its own window.

The main code file contains static arrays of documents and markers, and an array of tag pairs that points into the other arrays. A pointer to an element in the tag pairs array is passed as the client data.

XDhelp_link() calls the appropriate routines to communicate with FrameMaker to display the document as requested. Other implementations of *XDhelp_link()* communicate with different help systems. There are libraries supplied for integrating with Netscape and with Sun WorkShop Visual Help.

CHAPTER 22

Internationalization

Introduction

Sun WorkShop Visual helps you to make use of the special features that X and Motif provide to assist in developing applications that can be used in different languages. Although internationalization is a complex subject, this chapter points you in the direction of those areas that you will need to examine in order to fully internationalize your software.

In order to type text of languages other than your own, you will need an X server which supports that language. You can find out if this is the case by checking whether the *locale* is present on your system. See “Locale Name” on page 617 for some hints on how to find out.

What Is the Problem?

If your application is to be used by speakers of another language (even if that other language is *British* English rather than *American* English) you cannot assume that they understand your language and conventions. Apart from the variation in language, the use and style of addresses, date and time formats, personal names, even paper sizes may be very different from yours.

Scripts, in particular, can constitute a vast difference in the appearance and behavior of your application. Some languages, such as Korean, Japanese and Chinese, are ideographic. They do not have an alphabet as understood in Roman languages. Each word is represented in a graphical form. The fonts used to display such a script need to be able to handle thousands of these characters. This problem is discussed further in “Creating International Text” on page 620.

Other languages, such as Hebrew and Thai are alphabetic but still totally unlike Roman scripts. Some letters change according to context. There are often no spaces to distinguish words and some languages are written right to left while others are written left to right.

Even languages which appear to be very similar, such as French, English and German, involve differences in the symbols (or diacritics) used with letters and in the way in which letters are changed from lowercase to uppercase and vice versa.

X11 does assist you in addressing these problems by providing FontSets. These are explained in “Font Sets” on page 617. Also explained in this chapter is the mechanism by which *input methods*, required by non-alphabetic scripts, can be used in your application.

Although Japanese is used in most examples in this chapter, the principles are the same for any language.

It must be stressed here that this chapter aims to introduce you to the large area of internationalization and to explain how this is supported by Sun WorkShop Visual. You are strongly advised to refer to additional documentation for more in-depth information. This includes system documentation, Motif documentation and, where necessary, books such as those listed in Appendix E, “Further Reading”, starting on page 885.

Locale

Before you begin to address the issue of internationalization with regard to your application, you will need to understand how to set up your environment for a particular location. This involves an understanding of *locales*. A locale is an ANSI-C concept. It is a name that is used to identify a set of local information. For example, the locale might be set to be “en_UK” to denote that the location is the U.K. with English as the language, “en_US” for English as used in the U.S., or “ja” for Japanese. The setting of a locale makes certain C library functions operate in different ways, e.g. defining the sorting order, or date format.

Locale Name

You need to know the system name of the locale you are wishing to use and then set the LANG environment variable correspondingly so that applications can work out which language they should be displaying. Note that the name of the locale may differ depending on which system you are using. For a list of locale names supported by your UNIX system, look in the directory `/usr/lib/locale`. This will contain a directory for every language available. The directory names are used as the locale name, which is also the name you should use for the LANG environment variable. Type:

```
    locale -a
```

to list all locales currently installed on your system.

Specifying the Locale

To run Sun WorkShop Visual with internationalization support, you must set an appropriate locale. Do this by setting the *LANG* environment variable *before running your X server*, for example:

```
setenv LANG ja
```

This tells X to use Japanese for its display fonts and keyboard input, if Japanese is installed on your system.The value of the environment variable may be different on your system. See the *Locale Name* section above for more information.

For a language such as Japanese, which uses an input method *in-between* the keyboard and a text widget, you must make sure that you are running the appropriate version of Sun WorkShop Visual (in this case Japanese). This is because it is the presence of the Japanese fonts, in combination with the LANG environment variable, which indicates to the X server that an input method is required.

Font Sets

Part of a locale’s definition includes a specification of the font encodings required in order to display all possible text. For example, the Japanese locale requires that fonts with codesets ISO 8859-1, JIS X0208-1983 and JIS X0201-1976 be present in order to display all text in the Japanese language because Japanese contains Roman characters, Japanese alphabet characters (kana) and ideographic characters (Kanji). A FontSet is a collection of Fonts that contains all the required character sets for the current locale.

FontLists

The FontList is a collection of Fonts which is used for drawing labels in multiple fonts and styles. FontLists are specific to Motif whereas FontSets are defined by X.

For example, in order to specify that a string contains both bold and italic, you create a *compound string* with markers indicating which font (bold or italic) to use where. This is easily achieved in the Sun WorkShop Visual XmString editor (see “Compound Strings” on page 163). To do so, however, you need to define a Font Object in the Font Editor (see “Setting Fonts” on page 139) which contains a list of fonts - one bold and one italic. This is your FontList. Each Font in the FontList is tagged with the name you specify in the Font Editor. These tags are included in the compound string to tell the X server which font to use.

What Is the Connection Between FontSets and FontLists?

In order to incorporate FontSets, Motif has extended its definition of a FontList so that each entry in a FontList can be either a single Font (FontStruct) or a FontSet.

Sun WorkShop Visual simplifies this by presenting all the Fonts in a FontSet as straightforward members of the FontList. However, when you set the FontSet toggle in the Font Selection dialog, you are indicating to Sun WorkShop Visual that this Font is a member of a FontSet. Internally Sun WorkShop Visual gives all members of a FontSet the same FontList tag. Motif then collects together all Fonts with the same FontList tag as members of a FontSet.

See “Simple Font Objects” on page 144 for more information on font objects and “Creating a Complex Font Object” on page 164 for an explanation of FontLists.

Example of the Use of FontSets

The following exercise illustrates the use of FontSets.

The Japanese locale described above requires these three encodings:

- ISO 8859-1 for Latin characters
- JIS X0208-1983 for Kanji ideographic characters
- JIS X0201-1976 for Kana phonetic characters

1. **Pop up the Font selection dialog.**

2. **Enter a Font object `f1` and a tag name `t1`.**

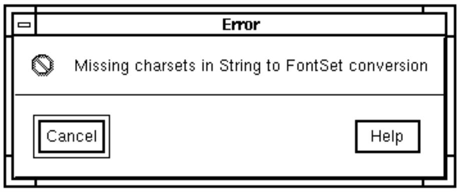
3. **Set the “Font set” toggle on.**

4. **Use the Reg Filter Menu to show all ISO 8859-1 fonts.**

5. **Select an appropriate ISO 8859-1 font and press bind.**

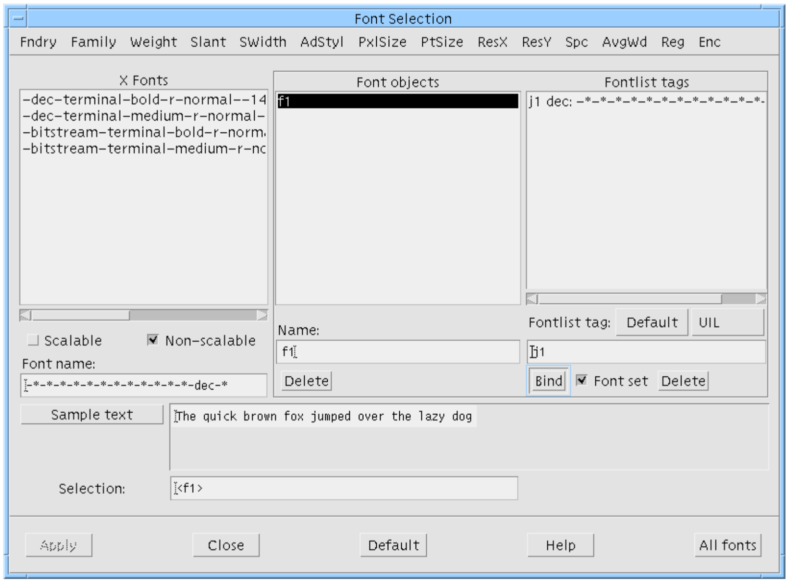
   Note that you should press the *Default* button as the tag name if you wish this FontSet to be used by default, otherwise you will have to provide code to make sure that the FontSet is used by your widget.

   At this point, Xlib warns you that you have a FontSet that does not have sufficient encodings charsets to display text in the current locale. This is correct, since you still have two fonts to do.



Missing charsets in String to FontSet conversion

FIGURE 22-1 Missing Charsets Warning



X Fonts:

-dec-terminal-bold-r-normal--14
-dec-terminal-medium-r-normal-
-bitstream-terminal-bold-r-norm
-bitstream-terminal-medium-r-nc

Font objects: f1

Fontlist tags: j1 dec: -*-*-*-*-*-*-*-*-*-*-*-*-

□ Scalable ☑ Non-scalable

Font name: -*-*-*-*-*-*-*-*-*-*-*-*-dec-*

Name: f1

Fontlist tag: j1

☑ Font set

The quick brown fox jumped over the lazy dog

Selection: <f1>

FIGURE 22-2 Initial Font in FontSet

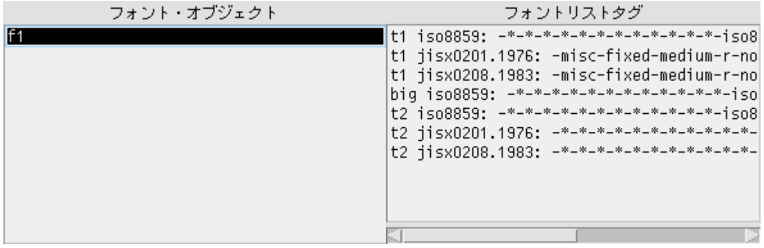
6. **Select a font with JIS X0208-1983 registry and bind it.**

   Xlib issues another warning of missing charsets.

7. **Select a font with JIS X0201-1976 registry and bind it.**

   This time Xlib does not warn you because the FontSet is now complete.

Font object entries can be deleted and re-bound regardless of whether or not they are part of a FontSet. A FontList can have many entries which can be either FontSets or simple fonts. To achieve this, specify different Fontlist tags and set the “Font set” toggle as appropriate. The following example shows a single font object that has three entries: *t1*, *big* and *t2*. *t1* and *t2* are complete FontSets for the ja locale; *big* is a simple font.



フォント・オブジェクト	フォントリストタグ
f1	t1 iso8859: -*-*-*-*-*-*-*-*-*-*-*-*-iso8
	t1 jisx0201.1976: -misc-fixed-medium-r-no
	t1 jisx0208.1983: -misc-fixed-medium-r-no
	big iso8859: -*-*-*-*-*-*-*-*-*-*-*-*-iso
	t2 iso8859: -*-*-*-*-*-*-*-*-*-*-*-*-iso8
	t2 jisx0201.1976: -*-*-*-*-*-*-*-*-*-*-*-
	t2 jisx0208.1983: -*-*-*-*-*-*-*-*-*-*-*-

FIGURE 22-3 A Font Object with Three Entries

Creating International Text

For a language such as Japanese, Chinese or Korean, more characters can be displayed than there are keys on the keyboard, or numbers that can be represented by 8 bits. In addition, text displayed by Japanese symbols must be mixed with Roman numerals and Roman text. These problems are addressed by coding the text in a special way. Instead of the familiar ASCII mapping, *multi byte* text strings are used in order to allow for the larger set of characters in these languages. Multiple characters in the text signify a single character from the matching font. The way in which one- and multi-byte characters are distinguished depends on the encoding. Such languages also have a one-to-many mapping of sound (as may be typed on the keyboard) and ideographic character. To resolve this, an *input method* is provided.

Languages such as Hebrew and Arabic also require an input method because the letters change according to the position in the word and vowels are written *around* the consonant.

If an input method is required and you are running an appropriately localized version of Sun WorkShop Visual, a special key sequence switches into using the input method when you enter text into a text field. The method converts the characters typed into corresponding characters in the target language.

Using an Input Method in Sun WorkShop Visual

To enter non-English text into any text input area in Sun WorkShop Visual first of all make sure that you fulfil the following requirements:

1. The LANG environment variable is set to the language type (locale) you require.

   See “Locale” on page 616 for more information.

2. You have an X server which supports the required language.

3. You have the appropriate version of Sun WorkShop Visual.

4. If the language specified needs an input method, it is present on the system and has either been started automatically or started by you.

The following example illustrates how to type text in a language which requires an input method. The language used is Japanese. If you are using CDE:

1. **Set the locale in the dialog box when you log in and then log in as usual.**

If you are not using CDE:

1. **Set the LANG environment variable to “ja”**

   If you are using OpenWindows you do not need to do anything else. If, however, you are using the Motif Window Manager, you should start the input method server by typing: **`htt`** at the command line prompt.

2. **Start your X server**

   Whenever a window containing a text field has the current focus, a *status line* appears at the bottom of the window. This is a text area displaying the current input mode.

Whether you are using CDE or not, the following instructions are the same:

3. **Start Sun WorkShop Visual.**

   You can read in a saved design or start a new one. Make sure that you have a Text widget.

4. **Select a Text widget, invoke its resource panel and select the “Label” field.**

5. **Type Ctrl-Space. Check you input method documentation for the correct key.**

   The status line changes to indicate that you may now type kana.

6. **Type Ctrl-W.**

   The kana is converted to Kanji and a *candidate list* is displayed allowing you to select the Kanji you want to enter into the text field.

7. **Click on an item in the candidate list and then press <Space> to send the selected Kanji to Sun WorkShop Visual.**

The procedure for other languages is similar - first, set the LANG environment variable, start your X server, then invoke Sun WorkShop Visual. For some languages and on some systems you may also need to start an input method or server. Refer to your system documentation to check whether this is necessary. The input method will appear whenever a window with a text area has the current focus.

For more information on international text on Sun platforms, see your Sun documentation.

Localized Input in Your Application

The previous section described how to enter non-ASCII text into Sun WorkShop Visual (for example into the label resource). This section describes how to enter localized text in your generated application. Note that the term “converted text” is used. This means any text on the screen which differs from that which appears on a *standard* keyboard.

How Text Is Converted

The X toolkit does most of the work for you. What you need to do is to indicate to the X toolkit that a particular text widget will be receiving non-ASCII text. This is accomplished by fulfilling these criteria:

1. The locale is set.

   See “Specifying the Locale” on page 617.

2. You have specified a font object for the text widget that includes a FontSet which matches the current locale.

If the second condition above is true, the text widget makes a request to the enclosing Shell to establish a connection to the input method. The X toolkit then takes over, diverting text input for that text widget to the input method and returning to the text widget the resulting converted text.

Setting up a Text Widget to Receive Converted Text

You can see from above that in order for a text widget in your application to use an input method, all you have to do is to make sure that the text widget has a FontList specified for it which includes a FontSet matching the locale *at creation time*. The simplest way to do this is to use Sun WorkShop Visual to specify a font object containing an appropriate FontSet and use that as the font resource for the widget. The following steps illustrate this:

1. **Create a dialog with a Text child.**
2. **Designate the Shell as an ApplicationShell.**
3. **Select the Text and pop up the resource panel.**
4. **Click on the Font button.**
5. **Specify a font object with an appropriate FontSet**

   Refer to “Example of the Use of FontSets” on page 618 for a description of how to do this. Remember to give the FontSet fonts the “Default” tag. If you do not, the X server may not realize that an input method is required.
6. **Generate code. Compile.**
7. **Run the generated application on an X server with support for the language specified for the Text widget.**

Seeing the Input Method in the Dynamic Display

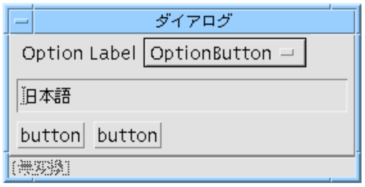
Although the above works correctly for your generated application, there is a drawback. Sun WorkShop Visual always creates the widget before resources are applied to it, even on reset or paste. Therefore, to see the input method working in the dynamic display window, you must force the font to be specified at creation time. This is best accomplished by setting the *textFontList* resource on a parent Bulletin Board or Shell widget. Obviously, this has the disadvantage that all the child text widgets have the input method.

To see a text widget working with an input method from within Sun WorkShop Visual, use the following steps.

1. **Create a dialog with a Form child.**
2. **Designate the Shell as an ApplicationShell.**
3. **Select the Form and pop up the “Fonts” page of its resource panel.**

4. **Click on the “Text font” button.**
5. **Specify a font object with an appropriate FontSet**

   Refer to “Example of the Use of FontSets” on page 618 for a description of how to do this. Remember to give the FontSet fonts the “Default” tag. If you do not, the X server may not realize that an input method is required.
6. **Create a Text widget child.**
7. **Add other widgets as required.**



日本語

FIGURE 22-4 A Simple Dialog with Input Method Attached

The default *main()* program includes a call to *XtSetLanguageProc()*, which initializes the locale handling routines. If you are writing your own *main()* program, you must insert a call to this routine if you are using any internationalization features. Add the following line to the beginning of your *main()* routine:

```
XtSetLanguageProc ( (XtAppContext) 0, (XtLanguageProc) 0, (XtPointer)
0 );
```

Setting the Application Font Resource

When using international text in Sun WorkShop Visual the text will not appear correctly in the dynamic display or generated application unless the appropriate FontList resource is set for the language you are using.

A useful way to implement the correct FontList settings on an application-wide basis is to use loose bindings to declare a FontList resource binding as in the following example:

Note – The line breaks in the following example have been inserted for clarity - if you are editing a resource file you must make sure that there are no line breaks, otherwise the resource setting will be ignored.

```
XApplication*FontList:
     -misc-fixed-medium-r-normal-*-14-130-75-75-c-140-
           jisx0208.1983-0;
     -*-*-medium-r-normal-*-14-130-75-75-c-70-jisx0201.1976-0;
     -misc-fixed-medium-r-normal-*-14-110-100-100
           -c-70-iso8859-1:default
```

where `XApplication` is the class name of your application. This example sets some Japanese fonts. The above example resource can be found in $VISUROOT/src/examples/loose_bindings/ja_fontlist.res and can be loaded into your design via the loose bindings dialog.

See "Loose Bindings" on page 86 for details on how to do this.

Using Eight-Bit Characters in Shell Titles

If you intend to use a character with the eighth bit set in the title string of a Shell widget, you must make sure that the *titleEncoding* resource is set to "`STRING`". You can do this from within Sun WorkShop Visual by specifying a loose binding which should look like this:

```
XApplication*titleEncoding:STRING
```

(where `XApplication` is the class name of your application.)

Or this:

```
*titleEncoding:STRING
```

depending on whether you wish the resource binding to apply to all applications (as in the second example) or just your own application.

See "Loose Bindings" on page 86 for details on how to do this.

Alternatively, if you prefer to "hard code" the setting of this resource in your design, you can set the pre-manage prelude for the main Shell as follows:

```
XtSetArg ( al[ac], XmNtitleEncoding, XA_STRING ); ac++;
```

See "Shell Pre-manage Prelude" on page 245 for details on adding a pre-manage prelude to a Shell widget.

Unsupported Locales

"Using an Input Method in Sun WorkShop Visual" on page 621 describes how you can specify a locale using the LANG environment variable. Providing you are using an appropriate version of Sun WorkShop Visual, you may then use the language of the locale for the strings in the user interface you are designing. If you have set your LANG environment variable to a locale which is not supported by your version of Sun WorkShop Visual, a message is displayed on startup informing you that your LANG setting is unsupported and that it is being coerced to "C". This is the default and is US English. If your Sun WorkShop Visual expects a Latin script[1], it will display and function correctly. The only area where there could be a problem is in strings in the generated code. If the following are true, you will be able to add labels, text strings etc. in the locale you specified:

- The locale you specified for LANG uses a Latin character set
- The appropriate font or fonts are available
- The language of the specified locale does not require special processing (i.e. needs an input method or does not display left to right)

If, however, any of the above is not true, you will probably not be able to enter or display strings in the specified language.

[1] The term "Latin script" here refers to the ISO Latin 1 character set.

CHAPTER 23

User-Defined Widgets

Introduction

Sun WorkShop Visual is pre-configured with the Motif widget set and can be extended to support widgets from any other source in addition to the default set. Widgets added to Sun WorkShop Visual are called *user-defined widgets*. They appear in the Sun WorkShop Visual widget palette and users can create them, set their resources and generate code for designs that include them.

Sun WorkShop Visual comes with a utility, visu_config, which helps you provide the information Sun WorkShop Visual needs to support user-defined widgets. You specify which widgets you want to use and provide information about any nonstandard resource types defined by the widgets. visu_config generates three C files that serve as a bridge between Sun WorkShop Visual and the user-defined widgets.

After generating the visu_config files, you build a new version of Sun WorkShop Visual from the following components:

- The Sun WorkShop Visual object file, `visu`*.o*
- Object files or archive libraries containing the added widgets
- The code file generated by visu_config
- The config file generated by visu_config
- Bitmap files for widget icons (optional)
- Pixmap files for widget icons (optional)
- Handwritten code files containing any auxiliary functions (optional)

Icons are recommended but not required. Handwritten code is only required if you want to provide customized popup dialogs, or if you have widgets with special problems described in “Configuration Functions” on page 672.

Using Pre-Configured Integration Kits

This chapter describes the use of visu_config. Integrating widgets into Sun WorkShop Visual is a complicated process. visu_config makes this process easier but is still a complex tool to use. However, Sun WorkShop Visual is distributed with pre-configured integration kits for several widget sets. Look in the directory `user_widgets` in your Sun WorkShop Visual release directory for some of the supported widget integrations. Many more are available on request, so it is highly likely that the set you are interested in already has an integration kit that you can use. Contact your Sun WorkShop Visual supplier for details of the full set available. Each integration kit is supplied with detailed information on how to use it. This is contained in the `README` file.

You probably only need to use visu_config if you are integrating a proprietary widget set or you wish to construct a palette containing more than one widget set in addition to the standard Motif set.

Requirements

User-defined widgets must build and run against the X11 Release 5 or Release 6 X Toolkit Intrinsics. For each widget class you need the public header file and the object file that implements the widget class. Both of these are provided by your widget supplier. The object file may be in an archive library. You may also need the private header file for the widget class.

You need access to the standard UNIX development tools, including the C compiler, linker and *make*.

visu_config requires standard widget information such as the widget class description. You should therefore have the widget documentation on hand. If your widget has non-standard resource types, or if you want to supply customized popup dialogs, you may also need to refer to the header files or widget source code to get the required information. See “Getting Widget Information” on page 634.

Generating UIL

Normally you would generate C or C++ for third party widgets. You can, however, generate UIL. You may need to add extra information in order to tell Sun WorkShop Visual how to do this. This is described in “Generating UIL” on page 679.

Generating Java Code

Sun WorkShop Visual generates Java code for user-defined widgets in your design by using a specially configured resource file which provides a mapping from user widget to Java component. All integration kits supplied with Sun WorkShop Visual (see “Using Pre-Configured Integration Kits” on page 628) include this resource file. The resource file is:

```
$VISUROOT/user_widgets/<USER_WIDGET_SET>/app-defaults/visu
```

where VISUROOT is your Sun WorkShop Visual installation directory and <USER_WIDGET_SET> is the name of the widget set that you are using - for example, XRT or Athena.

If you have created (or intend to create) your own integration, you can supply your own resource file. There are four resources relevant to Java code generation:

```
visu*xw_<WidgetClassName>.javaClassName
visu*xw_<WidgetClassName>.java10ClassName
visu*xw_<WidgetClassName>.java11ClassName
visu*xw_<WidgetClassName>.javaSwingClassName
```

The following shows one example of this:

```
visu*xw_XrtStringField.javaClassName: TextArea
visu*xw_XrtStringField.java10ClassName: TextArea
visu*xw_XrtStringField.java11ClassName: TextArea
visu*xw_XrtStringField.javaSwingClassName: JTextArea
```

Sun WorkShop Visual uses “javaClassName” if no more specific Java resource is found for the given version of Java being generated. In the example above, you do not need to supply values for “java10ClassName” and “java11ClassName” because they are the same as the default.

If a widget cannot be found in the resource file, Sun WorkShop Visual will substitute any container widget for the Java Panel class and will use the Java Canvas class for any other type of widget. For more information on Java code generation, see Chapter 10 ”Designing for Java“.

Generating MFC Code

Sun WorkShop Visual generates C++ and MFC code for user-defined widgets in your design using exactly the same mechanism as for Java code generation described in the section above. The following are recognised by Sun WorkShop Visual to configure third party component C++ classing:

```
visu*xw_<WidgetClass>.cppClassName
visu*xw_<WidgetClass>.motifClassName
visu*xw_<WidgetClass>.motifMfcClassName
visu*xw_<WidgetClass>.mfcClassName
```

Here is an example:

```
visu*xw_XtXrtLabel.mfcClassName: CStatic
```

The cppClassName is the default if a particular flavor is not found in the file.

The following will generate class “MyClass” for all variants of C++ in the absence of an MFC, MotifXP, or Motif specific information:

```
visu*xw_SomeWidgetClass.cppClassName: MyClass
```

And this will generate “AnotherClass” for all variants of C++ except pure MFC:

```
visu*xw_SomeWidgetClass.cppClassName: AnotherClass
visu*xw_SomeWidgetClass.mfcClassName: AnotherMfcClass
```

Caveats

Since Sun WorkShop Visual’s dynamic display works by creating actual instances of the widget, any widget you build into Sun WorkShop Visual becomes part of the tool. If the widget doesn’t function as expected, Sun WorkShop Visual may fail when the user adds the widget to a design. Any memory leaks in the widget can affect Sun WorkShop Visual and may cause a gradual degradation of performance or a core dump. Even widely-used widgets from standard vendors can have problems. You should therefore test widgets thoroughly before adding them to Sun WorkShop Visual.

Prerequisites

To configure Sun WorkShop Visual, you need some knowledge of C and an understanding of common UNIX development tools such as *make*. You don't have to be an expert on X but you need some knowledge of X and the X Toolkit Intrinsics. visu_config requires you to supply information about the widget, such as the widget class pointer and symbolic constants representing resource types. For suggestions on how to get the required information from the widget documentation or source code, see "Getting Widget Information" on page 634.

How Sun WorkShop Visual Works

Before you start configuring a widget, it is helpful to understand how Sun WorkShop Visual uses widgets in the dynamic display. This section describes some aspects of how Sun WorkShop Visual works internally.

Creating Widgets

Sun WorkShop Visual builds its dynamic display with real widgets, not simulations. It creates widgets by passing the widget class pointer to *XtCreateWidget()*. The widget class pointer also gives access to information about the widget's resources and their types so that Sun WorkShop Visual can build a resource panel for the widget.

The order in which widgets are created and managed in the dynamic display is different from the typical order of operations in an application. When the user clicks on an icon to add a widget to the hierarchy, Sun WorkShop Visual creates an instance of the widget, realizes it and manages it before any resources are set. When Sun WorkShop Visual reads a hierarchy from a file, however, the hierarchy is created from the bottom up. Each widget is first created, its resources are set and finally it is managed. In either case, create-only resources can't be set ordinarily in the dynamic display since Sun WorkShop Visual creates widgets before setting resources.

Some widgets, such as the Athena Form widget, require resources to be set at creation time or don't behave well when managed without children. In these cases, you can customize Sun WorkShop Visual's procedure for adding the widget to the hierarchy by specifying a Realize function in visu_config. (For more information, see "Realize Function" on page 673.

Note – Neither of these issues affect the generated code. Sun WorkShop Visual generates code that creates the hierarchy from the bottom up and sets all resources at creation time.

Highlighting the Selected Widget

When the user selects a widget in the tree, Sun WorkShop Visual highlights that widget's icon in the tree. It also highlights the widget itself in the dynamic display by swapping the widget's foreground and background colors. Highlighting the widget may cause problems if the widget has a create-only foreground resource. In this case, you can disable foreground swapping on the Widget Edit Dialog, as described in "Widget Attributes" on page 640.

Preventing Invalid Hierarchies

Sun WorkShop Visual prevents invalid hierarchies by disabling all palette icons for widgets that are not valid children for the selected widget. Also, when a new widget is added to the hierarchy, Sun WorkShop Visual doesn't automatically select it if it can't have children.

For user-defined widgets, Sun WorkShop Visual looks at the widget's superclasses to determine valid hierarchies. You can also specify configuration functions to customize a widget's requirements for valid child and parent widgets. For more information, see "Configuration Functions" on page 672.

Building the Resource Panel

Sun WorkShop Visual builds a resource panel for the widget based on resource names and resource types in the widget class record. Resources inherited from a known superclass, such as the Core widget or a Motif parent class, are left off the resource panel and can be set on the resource panel for the superclass.

Sun WorkShop Visual automatically assigns resources of standard types to appropriate pages on the resource panel. They can however be explicitly assigned to other pages if required. Most widget resources fall into one of the standard categories; for a summary, see “Resources” on page 645.

When the resource panel is displayed, Sun WorkShop Visual uses *XtGetValues()* to get the current values of all resources for the widget. All resources except enumerations are converted to text strings and displayed in text fields on the resource panel. The user can set the resource by editing a text string. In some cases, such as fonts, colors and callbacks, the user can supply text indirectly through a popup dialog. For resources that don’t already have a popup dialog in Sun WorkShop Visual, you can supply a popup; see “Popups” on page 657.

Setting Resources

When the user applies a new resource value, Sun WorkShop Visual converts the text string to a value and applies it to the widget with *XtSetValues()*. It then immediately calls *XtGetValues()* to retrieve all resource values for the widget, converts the values back to text and displays them in the resource panel. This procedure shows immediately whether the toolkit accepted the new value and whether the new value caused other resources to change.

A function called a *resource converter* is used to translate a text string to a resource value and back. For standard resource types, the converter functions are built in and you don’t have to do anything in visu_config. If your widget has a resource of a non-standard type, visu_config lets you specify information about the converter function. For details, see “Converters” on page 655. If you don’t provide this information, the user can type a text string to set the resource in the generated code or resource file but Sun WorkShop Visual can’t set it in the dynamic display because it can’t convert the text string to a value.

For enumeration resources, Sun WorkShop Visual builds an option menu which, by default, is placed on the “Settings” page of the resource panel. Sun WorkShop Visual can handle Boolean enumeration resources for user-defined widgets. For other enumerations, you must provide a list of valid values in visu_config so that Sun WorkShop Visual can build an option menu. For details, see “Enumerations” on page 650.

Saving Designs and Code Generation

Saving designs and parsing save files is straightforward. For resource settings, Sun WorkShop Visual writes the resource name and the text representation of its value to the *.xd* file. The resource name and value are saved as they appear on the resource panel.

When it generates code, Sun WorkShop Visual uses two versions of every resource name. In the generated X resource file, resources are identified by a *name* such as *label*. Sun WorkShop Visual gets resource names from the widget class record. In generated code, resources are identified by a *defined name* such as *XtNlabel*. Sun WorkShop Visual constructs the defined name by adding an *XtN* prefix to the resource name. If your widget doesn't follow this naming convention, visu_config lets you specify a function to construct the defined name. For details, see "Configuration Functions" on page 672.

For enumerations, Sun WorkShop Visual also uses two versions of each possible value: a *resource file symbol* such as *center* and a *code symbol* such as *XtJustifyCenter*. When you configure an enumeration, you must provide both versions of each value.

When Sun WorkShop Visual generates code for any widget class, it generates an *#include* for that class's public header file. visu_config lets you specify an *#include* file for each user-defined widget class.

Getting Widget Information

To configure your widget, you may have to supply one or more variable names and symbolic constants from the widget code. This section summarizes the information you may have to provide. Most of the information you need should be available in the documentation for the widget. If not, you can get it from the widget's public and private header file, from the widget source code (if available), or from your widget supplier's technical support service.

In the following paragraphs we mention naming conventions that are observed by many widget suppliers. However, naming conventions are not invariable rules. Always check the names in the documentation or source code.

The Widget Class Pointer

When you add a widget class in visu_config, you need to supply the *widget class pointer*. The widget class pointer is the name of a pointer variable of type *WidgetClass*. This pointer gives access to a structure containing information about the widget class, including a list of resources and their types. Sun WorkShop Visual uses this information to build a resource panel for the widget class. By convention, widget class pointers have names of the form *<classname>WidgetClass*.

If you cannot find the widget class pointer in the documentation, look in the public header file for a line such as:

```
externalref WidgetClass myWidgetClass
```

or

```
extern WidgetClass myWidgetClass
```

In either case, the widget class pointer is *myWidgetClass*.

Resource Information

The resource *name* is a character string used to identify the resource in generated X resource files and on the resource panel. Sun WorkShop Visual gets this name directly from the widget class record and so you don't need to supply it. This string is usually a straightforward name without a prefix, such as *label*.

The *defined name* is a symbolic constant used to identify the resource name in source code. By convention, the defined name has the form <*Prefix*>*N*<*name*>, where <*name*> is the resource name. To find defined names, look in the widget documentation, or look in the public header file for *#define* directives. For example, the following lines from the Athena Form header file identify the defined names *XtNtop* and *XtNbottom*:

```
#define XtNtop "top"
#define XtNbottom "bottom"
```

Non-Standard Resource Types

To configure resources of non-standard types, you need to know the *resource type*. This is not a type such as *unsigned char* but a symbolic constant defined as a string by which the widget class knows the resource type. By convention, resource types have the form <*Prefix*>*R*<*Type*>. For non-standard resource types, especially enumerations, <*Type*> may be the same as the resource name.

If the documentation for your widget gives a resource type as *foo*, look for a line like the following in the public header file:

```
#define XtRFoo "foo"
```

In this example, you would enter *XtRFoo* whenever visu_config asks for a resource type. The resource type can also be found in the source code for the widget. The following structure defines a resource whose resource type is *XtRFoo*. Note that you can also get the resource's defined name, *XtNfoo*, from this structure.

```
{
    XtNfoo, XtCFoo, XtRFoo,
    sizeof(foo), XtOffset( FooWidget, foo),
    XtRImmediate, (XtPointer) NULL,
}
```

Non-Standard Enumerations

If your widget has non-standard enumeration resources, you need to specify a list of possible values. In some cases you may have to read the source code to get the names you need. For instructions, see “Enumerations” on page 650.

visu_config - the Main Dialog

Run visu_config:

```
    visu_config
```

The main dialog shown in Figure 23-1 is displayed.

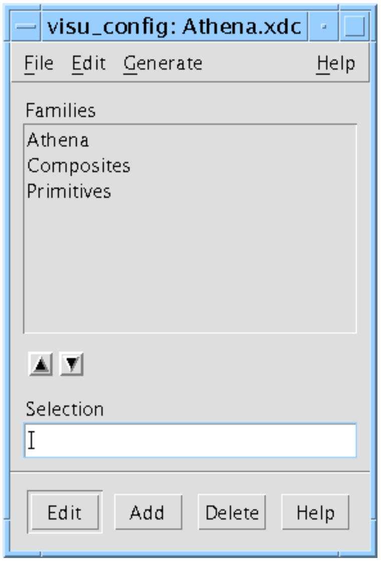


FIGURE 23-1 The Main visu_config Dialog

Menu Commands

The visu_config File Menu has options to save and read files containing your configuration data. By convention, these files have the suffix *.xdc*. Use “Open” to open an existing widget specification file; “Read” to merge another file with the one you are currently editing; “Save” and “Save As...” to save your file and “New” to clear the editing area.

The Edit Menu is used to display the Stop List dialog which lets you remove selected Motif widgets from the Sun WorkShop Visual palette. This is discussed in “Motif Widgets Stop List” on page 666.

The Generate Menu contains options to generate the two code files needed to build Sun WorkShop Visual with the added widgets. For information on generating code from visu_config and building Sun WorkShop Visual, see “Generating and Compiling Code” on page 667.

Families

The main dialog displays a list of *families*. Families are groups of widgets that are displayed together in the widget palette. The list is empty when you start the program. Figure 23-1 shows the dialog after loading the *Athena.xdc* file supplied with Sun WorkShop Visual. We recommend that you open this file and inspect it as you read.

You can organize user-defined widgets into families in any way. Grouping widgets into families has two purposes. First, it keeps the Sun WorkShop Visual widget palette to a reasonable size. At any given time, Sun WorkShop Visual displays the icons for the default Motif widgets plus one user-defined widget family. An option menu lets the user switch from one family to another as with the pages of a resource panel. Second, grouping widgets into families also makes it easy to generate versions of Sun WorkShop Visual with different sets of families. At code generation time, you can select any group of families from your list. This lets you customize Sun WorkShop Visual to support users with different needs and skill levels.

Editing the Family List

To add a new family to the list, type a name for the family in the “Selection” field, then click on “Add”. The family can have any name you choose. It is used to identify the family in visu_config and in the option menu in the Sun WorkShop Visual widget palette.

To delete a family, select it in the list and click on “Delete”. To reorder the list, select a family and use the arrow buttons to move it up and down. The order of the list determines the order of the items in the option menu in the Sun WorkShop Visual widget palette.

Suggestions for Organizing Families

You can include the same widget class in more than one family. For example, in *Athena.xdc*, the family named “Athena” contains all the Athena widgets and each of the two smaller families, “Composites” and “Primitives”, contains a subset of the Athena group. When you generate code from visu_config, you can decide how you want the widget palette to appear. You can either use the large family to display all the Athena widgets on the palette at the same time, or use either of the two smaller families to display a subset of the Athena widgets.

You might want to include a frequently-used widget in more than one family so that the user has access to it at all times regardless of what page of the palette is displayed. To do this, however, you have to enter and maintain two separate copies of the widget configuration information and you should test the icon separately on each page of the palette.

When you generate code from visu_config, you can select any group of families from the currently open file but you can’t select families from other files. To configure Sun WorkShop Visual with widget families from multiple *.xdc* files, use the “Read” option to merge the files before generating code.

Adding and Editing Widgets In a Family

To add or configure widgets in any family, select the family and then click on “Edit” to display the Family Edit dialog. The Family Edit Dialog lets you:

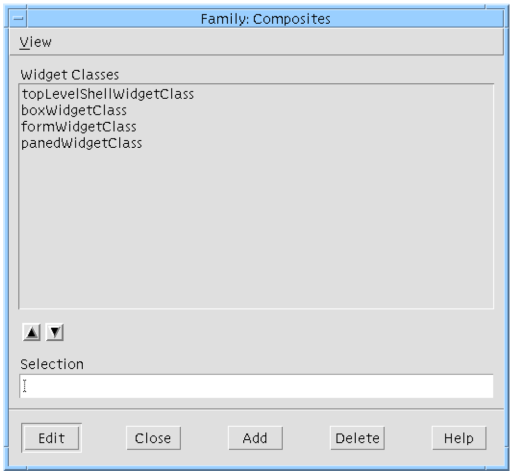
- Add or delete a widget class in the selected family
- Edit the specification for a widget class in the selected family
- Specify instructions for handling non-standard resource types
- Specify a popup dialog for any resource

The Family Edit dialog has several pages, which you can select from the View Menu. The name of the currently selected family is displayed in the dialog’s title bar.

For details about the Family Edit dialog, see the following sections.

Widget Classes

To display a list of widget classes in this family, select “Widgets” from the View Menu. Figure 23-2 shows the “Widgets” page for the Athena Composites family.



topLevelShellWidgetClass
boxWidgetClass
formWidgetClass
panedWidgetClass

FIGURE 23-2 The “Widgets” Page of the Family Edit Dialog

Adding a Widget Class

To add a new widget class to the family, enter the widget class pointer in the “Selection” field and click on “Add”. To complete the process, specify attributes for the class as described in the *Widget Attributes* section below.

Editing the Widget Class List

To delete a widget class from the family, select it in the list and click on “Delete”. To reorder the list, select an item and use the arrow buttons to move it up or down. The order of the widget class list determines the order of widgets in the palette.

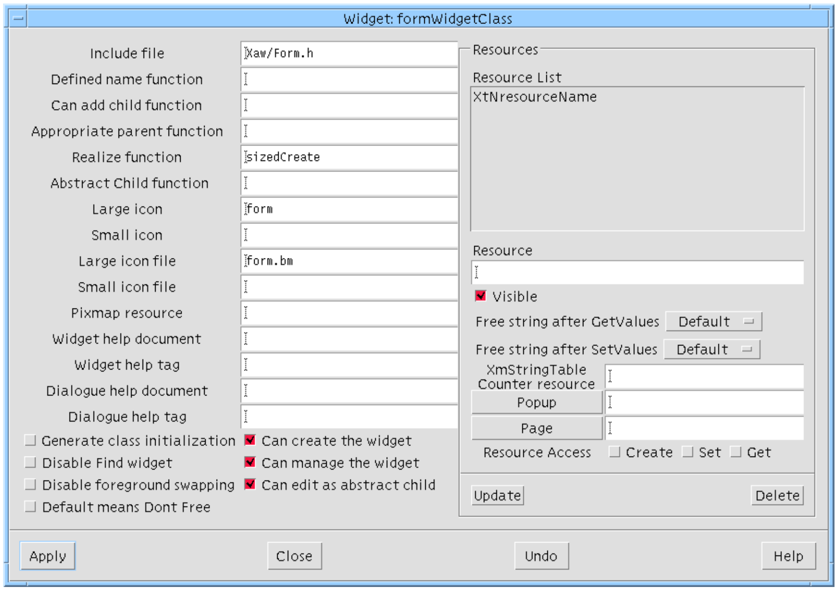
Widget Attributes

To specify attributes of a widget class, select the widget class in the Family Edit Dialog and click on “Edit”. This displays the Widget Edit Dialog, shown in Figure 23-3. The title bar displays the name of the widget class you are editing.

This section discusses the attributes set on the left side of the Widget Edit dialog. The right side is used to assign resources to existing or new pages, to specify custom popup dialogs for widget resources, and to override the default resource memory management. For details, see “Resources” on page 645.

Applying Changes

When you finish entering widget attributes, click on “Apply” to set the new values. “Undo” reverts to the last applied changes.



Widget: formWidgetClass

Include file	Xaw/Form.h
Defined name function	
Can add child function	
Appropriate parent function	
Realize function	sizedCreate
Abstract Child function	
Large icon	form
Small icon	
Large icon file	form.bm
Small icon file	
Pixmap resource	
Widget help document	
Widget help tag	
Dialogue help document	
Dialogue help tag	

☐ Generate class initialization
☒ Can create the widget
☐ Disable Find widget
☒ Can manage the widget
☐ Disable foreground swapping
☒ Can edit as abstract child
☐ Default means Dont Free

Resources

Resource List

XtNresourceName

Resource

☒ Visible

Free string after GetValues: Default

Free string after SetValues: Default

XmStringTable Counter resource

Resource Access ☐ Create ☐ Set ☐ Get

FIGURE 23-3 The Widget Edit Dialog with Attributes for Athena Form

Include File

In the “Include file” field specify the name of the public header file for the widget class. Enter the file name (usually relative to */usr/include*) without quotes or angle brackets. This file is included in two places: in the code file generated by visu_config and in application code generated by Sun WorkShop Visual.

Icons

An icon is an X pixmap or bitmap that represents a user-defined widget in the Sun WorkShop Visual widget palette and design hierarchy. For each widget, you can specify both a pixmap and a bitmap. Icon pixmaps are stored in separate files and specified via an entry in the Sun WorkShop Visual resource file.

Icon bitmaps are built into Sun WorkShop Visual and are used only if the pixmap resource is not set or the pixmap file cannot be found. If you don’t provide an icon in either bitmap or pixmap form, or the specified icon cannot be found, Sun WorkShop Visual displays a button with the widget class name in the widget palette and a crossed square icon in the hierarchy.

To reduce color usage it is recommended that you use the same color palette that Sun WorkShop Visual uses for its icons. Read the following file into the Pixmap editor by selecting “Read palette” from the “Palette” menu:

```
$VISUROOT/lib/palettes/icons.xpm
```

where VISUROOT is the install directory of your Sun WorkShop Visual. See “Read Palette” on page 155 for more information on reading palettes into the Pixmap editor.

Pixmap Resource

Use the “Pixmap resource” field to specify a name for the pixmap resource. After building Sun WorkShop Visual, you can set this resource to specify the pixmap file.

For example, if you specify *myWidgetPixmap* as the pixmap resource for a user-defined widget, you can specify a pixmap in the Sun WorkShop Visual resource file using the following entry:

```
visu.myWidgetPixmap: /usr/local/newwidget.xpm
```

This specifies */usr/local/newwidget.xpm* as the location of the XPM file. You can use either an absolute or a relative pathname. For details on how to use the pixmap resource, see “Palette Icons” on page 703.

Bitmap

To specify a built-in icon bitmap, you must provide two items of information: the name of the bitmap and the name of the corresponding bitmap file. To specify an icon for the large-screen (workstation) version of Sun WorkShop Visual, use the “Large icon” and “Large icon file” fields. For the small-screen (VGA) version, use the “Small icon” and “Small icon file” fields. You can provide an icon for either version of Sun WorkShop Visual, both, or neither.

The large-screen icon must be a 32 by 32 pixel X bitmap and the small-screen icon must be 20 by 20 pixels. You can create icon bitmaps using a tool such as the X bitmap utility. Pixmaps cannot be used for this purpose.

Enter the bitmap name in the “Large icon” or “Small icon” field. The bitmap name is defined in the first line of the bitmap file, as shown below:

```
#define <bitmap_name>_width 32
```

Enter the icon file name in the “Large icon file” or “Small icon file” field without quotes or angle brackets. Because this file is included when you build Sun WorkShop Visual, your Sun WorkShop Visual makefile must reference (-I) the directory where it is stored. This file does not have to be available to end users.

Help

These attributes let you specify on-line help that is displayed when the user invokes help for the widget on the palette or from the widget’s resource panel. For each help item, you specify a document (without any suffix) and a tag if the help for this widget is contained within a larger document. The Sun WorkShop Visual help system searches the path list specified by the visu.*helpDir* resource to locate the appropriate file. The file name must have a “.html” suffix. The tag refers to an HTML hypertext anchor within the document.

Configuration Functions

There are four fields for specifying configuration functions: a Defined Name function, a Can Add Child function, an Appropriate Parent function and a Realize function. These functions can be used to fine-tune the way Sun WorkShop Visual handles the widget in the dynamic display. If your widget uses a configuration function, type the name of the function in the corresponding text field.

The configuration functions perform the following tasks:

- *Defined Name function* – Translates resource names to appropriate defined names; required only if the widget doesn’t follow standard Motif naming conventions

- *Can Add Child function* – Determines whether you can add a widget of another class as a child of this widget
- *Appropriate Parent function* – Determines whether a widget of another class is an appropriate parent for this widget
- *Realize function* – Adds initialization steps when the widget is created in the dynamic display; required only if the widget cannot be created and managed satisfactorily without children, or requires resources to be set at creation time

The Can Add Child and Appropriate Parent functions are required only for widgets that have special requirements for valid hierarchies. Often these functions are not needed. If you do not supply them, Sun WorkShop Visual uses the rules for the first known ancestor of the widget class. For example, if a user-defined widget is derived from the Primitive class, Sun WorkShop Visual uses the rules for the Primitive class and does not let the user add children to the widget.

For full definitions, synopses and examples of configuration functions, see "Configuration Functions" on page 672. Note that many widgets do not require configuration functions.

The Miscellaneous Toggle Buttons

At the bottom of the dialog, there are a number of toggle buttons for turning on or off various options relating to the use of the widget. These are detailed in the following sub-sections. Note that the toggles in the "Resources" panel of the Widget Edit dialog are described separately in "Resources" on page 645.

Generate Class Initialization

When you create a widget for the first time, it usually does some initialization associated with the specific class of the widget (as opposed to the given widget instance).

With some widget sets, leaving this initialization until first create of a given widget may cause a problem; it is often better to explicitly initialize the widget class by hand before creating any of the widgets in the given widget set. This is the case where you have some implicit dependencies between the widgets. For example, you may have a Table widget which can have a Table Label widget as a child. However, some code in the Table may assume that the Label class has been initialized. In such a case, you would need to pre-initialize the Label class before you use a Table, even if you are not going to attach a Label into the Table.

If this toggle is set, Sun WorkShop Visual will generate the following line into your main module:

```
XtInitializeWidgetClass(widget_class)
```

before calling any creation code.

This toggle is set by default, since it does not cause a problem to pre-initialize the class, but, for some widgets at least, it can cause problems if you do not.

Disable Find Widget

Unset this toggle if you wish to prevent the user from using the Fast Find facility on this widget. You may wish to turn off this option if Fast Find does not interact well with your widget in the dynamic display.

Disable Foreground Swapping

Normally, Sun WorkShop Visual highlights the currently selected widget in the dynamic display. To disable highlighting for this widget class, turn on the “Disable Foreground Swapping” toggle. Do this only if the widget class has a foreground resource that cannot be set after widget creation time. This is required because Sun WorkShop Visual’s highlighting procedure can cause problems with such widgets. In all other cases, leave the toggle off.

Default Means Dont Free

This allows you to set the default way in which Sun WorkShop Visual handles the memory management of XmString and String (char *) widget resources. This topic is discussed in more detail, including the use of this toggle, in “Resource Memory Management” on page 664.

Can Create the Widget

When the “Can Create the Widget” toggle is on, users can build the widget directly into hierarchies. Leave this toggle on if you want the widget’s icon to appear in the palette.

Turn the toggle off to make the widget class an invisible superclass like the Core widget. If you turn the toggle off, the widget doesn’t appear in the widget palette and users can’t create instances of it directly. Sun WorkShop Visual creates a separate resource panel for the class. The resource panel can be accessed by any derived widget classes.

Can Manage the Widget

Not all widgets are GUI components. For example, the various Data Object classes in the INT ChartObject widget set are for representing components of a graph (Circles, Rectangles, Axes, etc.) in an MVC (Model View Controller) model. In this case, you would not render these objects directly but let the graph parent handle it all internally.

Sun WorkShop Visual does have some internal rules over what it can and cannot manage: it knows that managing anything that is an ObjectClass or derivative (as opposed to a WidgetClass) is forbidden by the toolkit - it will corrupt the ObjectClass if you do so. However, some widget sets have Data Objects derived via WidgetClass, even though these are not meant to be managed. Sun WorkShop Visual cannot be aware of these exceptions; its rules about what to manage would be wrong, both internally and in the generated code. Use this toggle, therefore, in those rare occasions when you need to stop Sun WorkShop Visual managing the given component.

Can Edit as Abstract Child

This refers to visu's ability to display, configure and generate code for the abstract children of composite widgets. The Motif ScrolledWindow widget is an example of a composite widget - the scrollbars are the abstract children. Turn this toggle off if your widget is a composite and you do not wish users to have any access to the abstract children. More information on accessing the abstract children of third party composite widgets in Sun WorkShop Visual is given in "Accessing Abstract Children" on page 669.

Resources

By default, when Sun WorkShop Visual creates the resource dialog for a widget, it gets the name and type of the resource directly from the widget code and builds a resource panel. It assigns each resource to a page of the resource panel (based on the resource type) and assigns a popup dialog for certain types. For example, resources of type *XtRPixel* are put on the Display page of the resource panel, with a button to pop up the Sun WorkShop Visual color editor.

You do not have to do anything to configure a resource unless you want to change this default behavior or unless the resource is not of a type listed in the table of standard types shown below.

Default Handling of Standard Resource Types

Resources of standard types are assigned to pages of the resource panel as shown in the following table:

TABLE 23-1 Sun WorkShop Visual Standard Resource Types

Resource Type Symbol	Value	Page
XmRDimension	“Dimension”	Margins
XmRFontStruct	“FontStruct”	Display
XmRHorizontalDimension	“HorizontalDimension”	Margins
XmRInt	“Int”	Margins
XmRPixel	“Pixel”	Display
XmRPixmap	“Pixmap”	Display
XmRPosition	“Position”	Margins
XmRPrimForegroundPixmap	“PrimForegroundPixmap”	Display
XmRShort	“Short”	Margins
XmRString	“String”	Display
XmRVerticalDimension	“VerticalDimension”	Margins
XmRWidget	“Widget”	Display
XmRXmString	“XmString”	Display
XtRBoolean	“Boolean”	Settings
XtRCallback	“Callback”	Callbacks
XtRUnsignedChar	“UnsignedChar”	Settings
XmRFontList	“FontList”	Display
XtRFontStruct	“FontStruct”	Settings
XmRXmStringTable	“XmStringTable”	Display

Changing Widget Attributes

The right side of the Widget Edit dialog lets you do the following things for individual widget resources:

- Inhibit the resource from appearing on the resource panel
- Assign the resource to a selected page of the resource panel
- Specify a popup dialog for setting the resource

- Control the memory management of the resource. This is described in "Resource Memory Management" on page 664.
- Control the "CSG" (Create / Set / Get) accessibility of a widget's resource.

To customize the attributes of a resource, type the resource name in the Resource field. This should be the defined name, such as *XtNcursorName*.

To remove the resource from the resource panel, turn off the "Visible" toggle.

To specify a different page of the resource panel, click on the "Page" button to display the current list of pages. The list is preset with the default Sun WorkShop Visual pages. To add a page, type the name of the new page in the "Page name" field and then click on "Update". To assign the current resource to a page, select the page in the list and then click on "Apply".

visu_config automatically invokes the Sun WorkShop Visual predefined popups for some types of resources, as shown in the following table. You don't have to specify a popup explicitly to use these popups for resources of these types:

TABLE 23-2 Standard Resource Popups

Resource Type	Popup
XmRFontList	Font selector
XtRFontStruct	Font selector
XmRPixel	Color selector
XmRPixmap	Pixmap editor
XmRPrimForegroundPixmap	Pixmap editor
XmRXmString	Compound String editor
XtRCallback	Callback dialog

You can specify a popup dialog for any resource. You can select one of the predefined popups or create your own. For more information, see "Popups" on page 657.

XmStringTable Counter Resource Text Field

Some resources come in pairs. When an array of XmStrings is passed to a Motif list, you also have to say how many strings are in the array. The following code, for example, causes your application to core dump:

```
XmString *xmstrings = ... ;
XtVaSetValues(list, XmNitems, xmstrings, NULL) ;
```

The array resource has no notion of an end-of-array marker, such as a null terminated list. You have to explicitly give the count, as in the following example:

```
int n = ... ;
XmString *xmstrings = ... ;
XtVaSetValues(list, XmNitems, xmstrings, XmNitemCount, n, NULL) ;
```

The “XmStringTable Counter resource” text field is for pairing together the array and array counter resources, so that X-Designer can use them together internally and also generate the right code with both resources used at once.

This is for both XmString array resources and for char ** array resources which each need a paired counter resource.

Create / Set / Get

There are three toggles controlling the accessibility of a widget’s resources. The toggles are labelled “Create”, “Set” and “Get”. These relate directly to “CSG” in the Motif documentation. Unsetting the “Set” toggle will make the resource read only in Sun WorkShop Visual and in the code generated by Sun WorkShop Visual. Unsetting the “Get” toggle means that you will not, in your code, be able to fetch the value of the resource. Unsetting “Create” means that you will have to create the resource yourself.

Documentation for third party widgets is often unreliable for “CSG” (Create Set Get), therefore the feature can be disabled by setting the resource:

```
visu.xwResourceAccessControl: false
```

Nonstandard Resource Types

By default, resources of types not in the table of Sun WorkShop Visual standard resource types are placed on the “Miscellaneous” page of the resource panel. The user can set them by typing strings into text fields. The strings are generated into the code exactly as the user types them and resource settings take effect when the generated code is compiled.

This default behavior has some disadvantages. Sun WorkShop Visual doesn’t recognize the resource type, so it cannot set the resource in the dynamic display. Enumeration values must be spelled and capitalized correctly (including any prefix) or the generated code will not compile.

visu_config offers several ways to refine the way Sun WorkShop Visual handles nonstandard resource types:

- You can specify an *alias* for a resource whose type behaves like one of the standard types. For example, if your resource is of a non-standard type that works the same as *XmRInt*, you can instruct Sun WorkShop Visual to treat it as *XmRInt*
- You can configure Sun WorkShop Visual to handle *enumerations* of types other than *XtRBoolean*. After you do this, the enumeration resource is placed on the "Settings" page of the resource panel from where the user can set it with an option menu. Misspellings are prevented and the resource is active in the dynamic display
- You can specify *converters* for resources of other types. The converter lets Sun WorkShop Visual set the resource in the dynamic display
- You can specify a *popup dialog* for any resource. For resources with popup dialogs, Sun WorkShop Visual creates a PushButton on the resource panel to invoke the dialog

The following sections give detailed instructions for these procedures.

Aliases

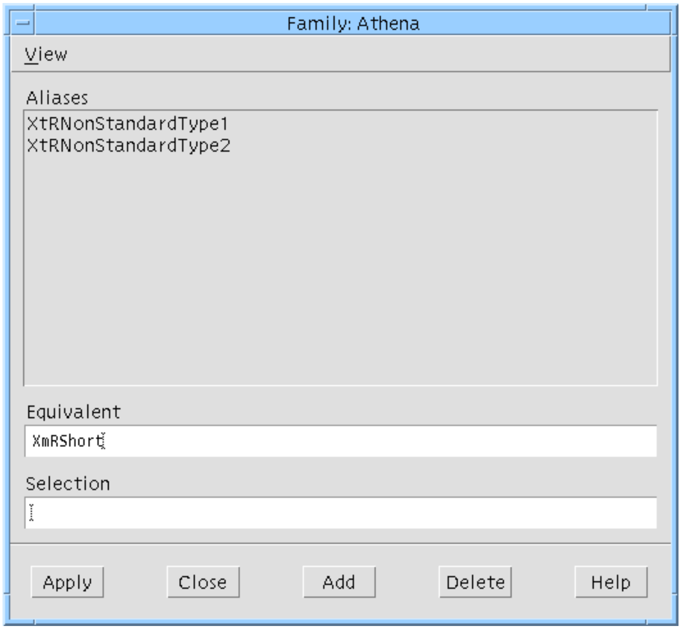
If your widget uses a resource type that is not in the standard list but has the same semantics as a standard type, you can tell visu_config that your type is an alias for the standard type. For example, if you define a type *XtRDegrees* as an integer from 0 to 359, you can set up an alias to specify that *XtRDegrees* is equivalent to *XmRShort*. The user can then set any *XtRDegrees* resource on the "Margins" page of the resource panel and see the results immediately in the dynamic display.

Requirements

The new resource type must have the same semantics as the standard type. Specifically, the process by which a text string is converted to a resource value and back again must be the same for both types.

Specifying an Alias

On the Family Edit Dialog, pull down the View Menu and select "Aliases" to display the dialog shown in Figure 23-4.



XtRNonStandardType1
XtRNonStandardType2

Equivalent: XmRShort

FIGURE 23-4 The “Aliases” Page of the Family Edit Dialog

To specify an alias, enter the non-standard resource type in the “Selection” field and the name of the corresponding standard in the “Equivalent” field. Click on “Apply” to register the alias.

Enumerations

Enumeration resources are resources with a fixed set of possible values. You can determine if a widget has enumeration resources by inspecting the documentation. If the documentation lists all possible values for a resource, it is an enumeration. Note that enumerations of type *XtRBoolean* are handled automatically and don’t require the configuration procedure described in this section.

By default, Sun WorkShop Visual treats all other enumerations as it treats any unknown resource type. It places them on the “Miscellaneous” page of the resource panel and the user can set them by typing a new value in a text field. The setting is passed to the generated code but has no effect in the dynamic display.

Use the instructions in this section to give Sun WorkShop Visual the information it needs to build an option menu for the resource on the “Settings” page of the resource panel and set the resource in the dynamic display.

Configuring an Enumeration

Select “Enumerations” from the View Menu in the Family Edit Dialog to display the “Enumerations” page. Figure 23-5 shows the list of enumerations for the Athena Primitives family.

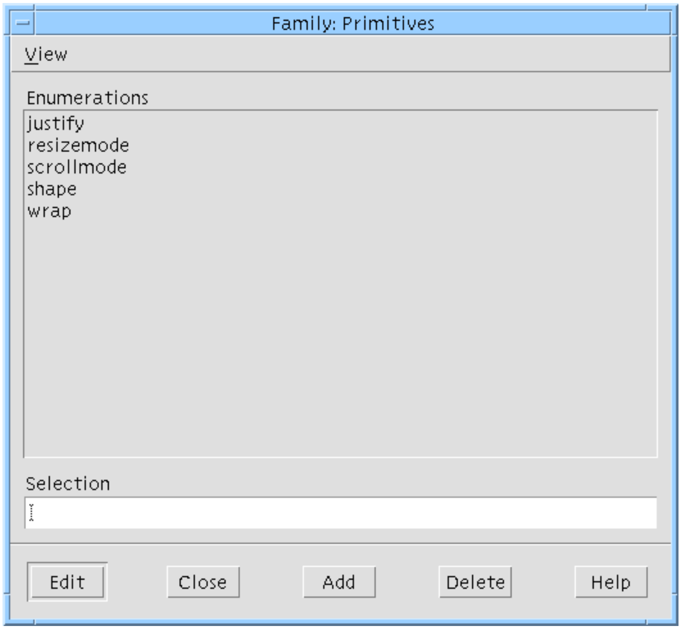


FIGURE 23-5 The “Enumerations” Page of the Family Edit Dialog

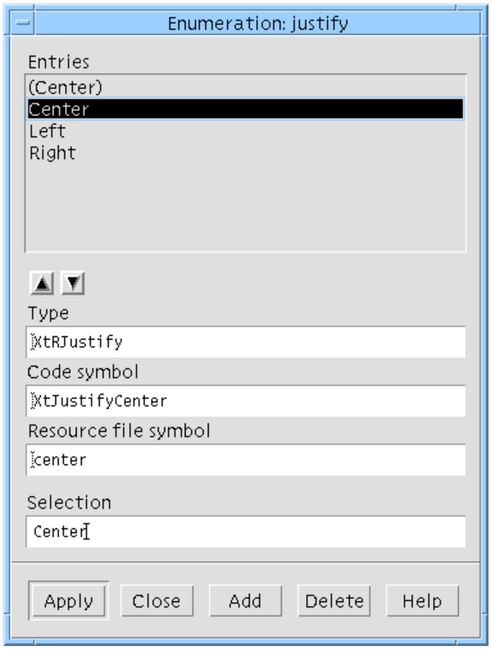
To add a new enumeration, enter a name in the “Selection” field and click on “Add”. This name is only for your convenience in visu_config. You can use the resource name from the widget documentation or any other string that helps you identify the resource. Since all enumerations for the family are kept together in one list, it may be useful to include the widget name as well as the resource name.

Configuring Enumeration Values

To complete the process, you need to provide the following information about the enumeration resource:

- The resource type
- A list of possible values
- The default value
- The code symbol for each value
- The resource file symbol for each value

Suggestions for getting the code symbol and resource file symbol are given at the end of this section. To configure an enumeration, select it in the list of enumerations and click on “Edit”. This displays the Enumerations Entry Dialog, shown in Figure 23-6. The title bar displays the name of the enumeration.



Entries
(Center)
Center
Left
Right

Type: XtRJustify

Code symbol: XtJustifyCenter

Resource file symbol: center

Selection: Center

FIGURE 23-6 The Enumerations Entry Dialog

Specify the resource type and configure each value as described below. When you finish, click on “Apply” to set the new values.

Specifying the Type

Enter the resource type in the “Type” field.

Specifying Values

To get a list of values for an enumeration, look in the widget documentation or the public header file. Make an entry in the “Entries” list for each possible value. Entries in the list are only used in the option menu on the resource panel and can be any names you want. Sun WorkShop Visual uses names that are meaningful to the user, such as *Horizontal* rather than *XmHORIZONTAL*.

For each value, type the name in the “Selection” field and click on “Add”. Note that omitting a value isn’t fatal but the user won’t be able to set that value.

Configuring Values

For each value, you must specify the *code symbol* and the *resource file symbol* in the Enumerations Entry dialog. The code symbol is a symbolic constant that denotes the value in generated code. The resource file symbol is a string that denotes the value in resource files.

To find the code symbol, look at the list of values in the widget documentation or the widget source code. Type the code symbol in the “Code symbol” field.

Type the resource file symbol for the value in the “Resource file symbol” field. Often, the resource file symbol is the code symbol, converted to lower case and stripped of any prefix, but this is not an invariable rule. For example, the resource file symbol for *XtJustifyCenter* is *center.*

The resource file symbol is not always listed in widget documentation. If it isn’t, you may be able to get it from an example resource file or from the widget supplier’s technical support service. If you have source code for the resource converter, you can get the resource file symbol from the code, as described in “Getting the Resource File Symbol” on page 654.

Specifying the Default Value

The first entry in the list of values on the “Enumerations” page is reserved for the default value. This entry is used to indicate a resource that is not set explicitly. The option menu should also contain an entry for the same value set explicitly. By convention, the default value is distinguished by putting its name in parentheses, as shown in the following list:

```
(Center)
Center
Left
Right
```

Specify the same code and resource symbols as for the corresponding explicit value.

Note that Sun WorkShop Visual builds one option menu for the each enumeration type. If your widget has multiple resources of the same enumeration type, they share an option menu. If the resources have different default values, a suggested approach is to enter a generic default value, *(Default),* on the option menu.

Enter code and resource symbols for any one of the possible default values. This does not result in errors in the generated code or when widgets are initially created in the dynamic display. The dynamic display may be incorrect if the user explicitly sets this resource and then explicitly requests the default value. However, any such problem disappears after the widget is reset.

Specifying Order of Entries

The order of entries in the list on the “Enumerations” page controls the order in which they appear in the option menu on the resource panel. Entries can be listed in any order as long as the default value is listed first. To move an entry to a different position, select it in the list and use the arrow buttons to move it up or down.

Getting the Resource File Symbol

This section explains how to get the resource file symbol you need to enter on the Enumerations Entry Dialog from the resource converter code for the widget. The resource converter is a function used to convert a string read from the resource file to the corresponding value. A simple converter may contain fragments like this:

```
if (StringsAreEqual (in_str, "vertical"))
    i = XmVERTICAL;
else if (StringsAreEqual (in_str, "horizontal"))
    i = XmHORIZONTAL;
```

In this example, the string *“horizontal”* is converted to the value *XmHORIZONTAL* and *“vertical”* is converted to *XmVERTICAL*. *horizontal* and *vertical* are the resource file symbols; *XmHORIZONTAL* and *XmVERTICAL* are the code symbols.

You may find it done more indirectly, as shown in the following code:

```
if (!haveQuarks) {
    XtQEhorizontal = XrmStringToQuark(XtEhorizontal);
    XtQEvertical = XrmStringToQuark(XtEvertical);
    haveQuarks = 1;
}
XmuCopyISOLatin1Lowered(lowerName, (char *)fromVal->addr);
q = XrmStringToQuark(lowerName);
if (q == XtQEhorizontal) {
    orient = XtorientHorizontal;
    done(&orient, XtOrientation);
    return;
}
if (q == XtQEvertical) {
    orient = XtorientVertical;
    done(&orient, XtOrientation);
    return;
}
```

In this code, the resource file symbols are represented by the symbolic constants *XtEhorizontal* and *XtEvertical*. To get the resource file symbols, *horizontal* and *vertical*, examine the related header files for lines such as:

```
#define XtEhorizontal "horizontal"
#define XtEvertical "vertical"
```

The code file symbols, *XtorientHorizontal* and *XtorientVertical*, are the end result of the conversion. Note that in this example there is an intermediate step: the strings are converted to quarks and the quarks are used for the comparison. The mechanics of the conversion procedure do not affect visu_config.

Converters

If your user-defined widget has resources of non-standard types other than enumerations, Sun WorkShop Visual places them on the “Miscellaneous” page of the resource panel by default. The resources can be allocated to other pages from the Widget Edit Dialog. The user can set these resources by typing a string into a text field. By default, the string is generated into the code or resource file but Sun WorkShop Visual doesn’t set it in the dynamic display. To make the resource work in the dynamic display, you can configure Sun WorkShop Visual with a *converter function* for this resource. The converter function is a function that converts a text string to a resource value.

To configure the converter, select “Converters” from the View Menu. This displays the page shown in Figure 23-7. The “Entries” list contains a list of resource types in this family for which converters have been specified.

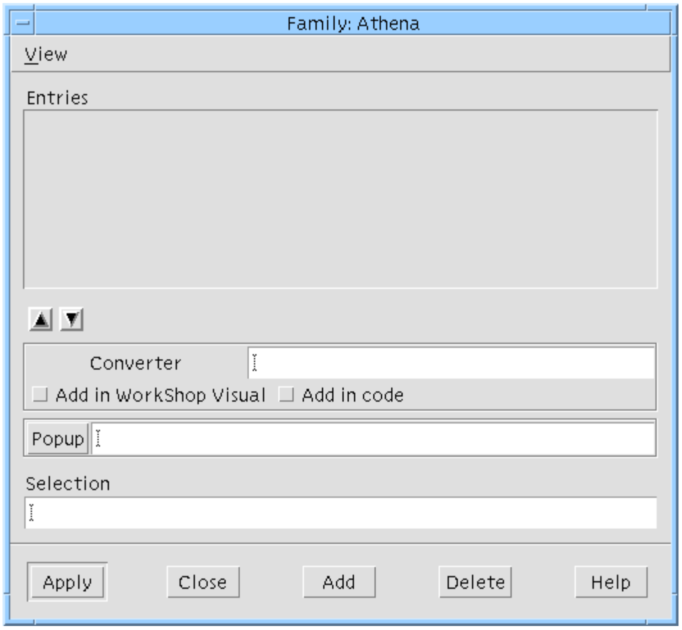


FIGURE 23-7 The “Converters” Page of the Family Edit Dialog

Resource Type

Enter the resource type in the “Selection” field and click on “Add”.

Converters Added Internally

Many widgets add their own converters internally when the class is initialized. If this is the case, all you have to do is add the resource type to the list on the “Converters” page. Adding the resource type to the list informs Sun WorkShop Visual that the converter is available; otherwise Sun WorkShop Visual doesn’t attempt to set the resource in the dynamic display.

To find out whether the widget class adds its own converter, look in the widget documentation or look for a call to *XtSetTypeConverter()* in the code for the widget’s Class Initialize method. If the converter is not added internally or if you are in doubt, instruct Sun WorkShop Visual to add the converter explicitly, as described below.

Converters Added Explicitly

If the widget doesn't add converters internally, you can instruct Sun WorkShop Visual to add them explicitly. To do this, specify the name of the converter function in the "Converter" box. The converter must be a function of type *XtTypeConverter*. Toggles are provided to add the converter explicitly either in Sun WorkShop Visual (for use in the dynamic display), in the generated code, or in both. In general, if you have to add the converter explicitly, both toggles should be on.

Popup Dialog

The "Popup" button lets you specify a popup dialog to be used for setting all resources of this type. For details, see the *Popups* section below.

Popups

Resource popups are dialogs used to set a resource, such as Sun WorkShop Visual's color and font selectors. For resources that have popups, Sun WorkShop Visual creates a button on the resource panel to invoke the popup, in addition to the usual text field. Figure 23-8 shows popup buttons on the Sun WorkShop Visual Core resource panel.

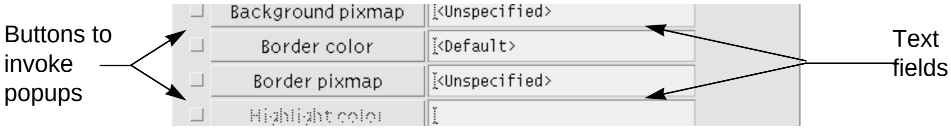
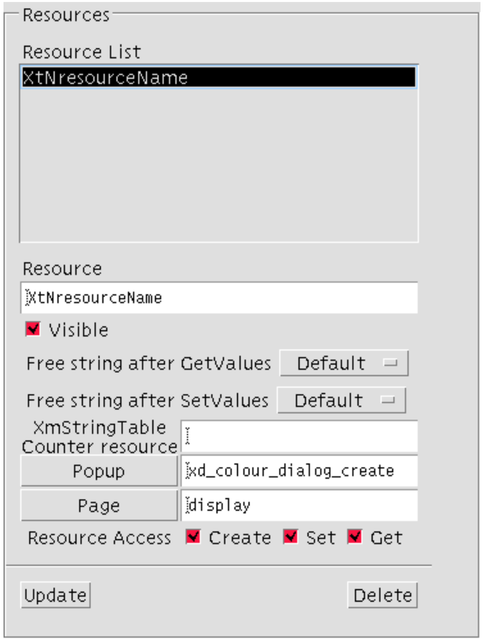


FIGURE 23-8 Popup Buttons on Core Resource Panel

Popups for Individual Resources

You can specify a popup dialog for any individual resource. To do this, use the right side of the Widget Edit Dialog, shown in Figure 23-9. Select the name of the resource in the list. If you haven't yet added the resource to the list, enter the defined name of the resource, such as *XtNresourceName*, in the "Resource" field and click on "Update".



Resources

Resource List

XtNresourceName

Resource: XtNresourceName

☑ Visible

Free string after GetValues: Default

Free string after SetValues: Default

XmStringTable Counter resource:

Popup: xd_colour_dialog_create

Page: display

Resource Access ☑ Create ☑ Set ☑ Get

FIGURE 23-9 Popup Portion of Widget Edit Dialog

This dialog uses resource names and not resource types. Therefore, you can specify different popups for different resources of the same type. For example, if you specify a popup dialog for a specific resource of type *XmRInt*, that popup is not displayed for other resources of that type.

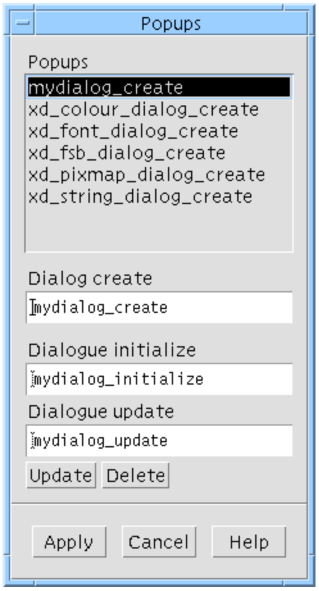
After you enter the resource name, click on the “Popup” button to display the Popups Dialog. Select a popup using the instructions in “The Popups Dialog” on page 659 section below.

Popups for Resource Types

If you specify a converter for a resource type, you can specify a popup dialog for that type. To do this, click on “Popups” on the “Converters” page of the Family Edit dialog. This displays the Popups Dialog. Select a popup using the instructions in the following section.

The Popups Dialog

The Popups Dialog is shown in Figure 23-10.



Popups
mydialog_create
xd_colour_dialog_create
xd_font_dialog_create
xd_fsb_dialog_create
xd_pixmap_dialog_create
xd_string_dialog_create

Dialog create: mydialog_create

Dialogue initialize: mydialog_initialize

Dialogue update: mydialog_update

FIGURE 23-10 The Popups Dialog

The Popups Dialog displays the current list of popups. The list is preset with the built-in Sun WorkShop Visual popup creation functions:

TABLE 23-3 Built-In Popups

Creation Function	**Popup**
xd_colour_dialog_create	Color selector
xd_font_dialog_create	Font selector
xd_fsb_dialog_create	File selection dialog
xd_pixmap_dialog_create	Pixmap editor
xd_string_dialog_create	Compound String editor

To apply an existing popup to the currently selected individual resource or converter type, select the popup creations function in the list and click on “Apply”. Note that all popup dialogs return the resource value in the form of a string such as *“red”* or “*<big_font>*”. Therefore they work for resources that are declared as strings but are used to specify fonts, colors, or filenames.

Custom Popup Dialogs

You can create your own popup dialog, add it to the list and apply it to any resource or converter type. Three functions are required for each popup dialog: a create function, an initialize function and an update function. Enter the name of each function in the appropriate text field and then click on "Update" to add the popup to the list.

Code Requirements

The popup functions are invoked at different points in Sun WorkShop Visual, as shown below:

TABLE 23-4

Function	Called
create function	When the dialog is first popped up.
initialize function	Each time the user clicks on the resource button.
update function	Each time a new widget is selected.

Add callbacks on widgets in your popup dialog to furnish hooks for additional functionality such as setting the new value, editing the value, accessing help and popping down the dialog. Your dialog should have at least the following standard callbacks:

TABLE 23-5

Callback	Functionality
Apply callback	Sets new resource value in source text widget.
Close callback	Pops down the dialog by unmanaging it.

The Text or TextField widget on the resource panel (called the *source text widget*) is used to pass information from the dialog to the resource panel. When the user sets a value on the popup dialog, the value should be converted to text and set into the source text widget. The user can then click on "Apply" on the resource panel to set the value in the dynamic display, just as if the text had been typed by hand.

You can build your dialog in Sun WorkShop Visual or code it by hand. The example at the end of this section shows a popup dialog that was built in Sun WorkShop Visual.

Create Function

The create function is called the first time the user clicks on the resource button to pop up the dialog. This function should create the widget hierarchy for the dialog.

```
void popup_create ( Widget parent )
```

The *parent* parameter is the Application Shell widget for Sun WorkShop Visual, to be used as the parent widget for the dialog. This parameter is required to call functions such as *XmCreateDialogShell()*. You can build popup dialogs in Sun WorkShop Visual and use Sun WorkShop Visual's generated creation procedure as the create function, or write a create function that calls it. In this case, pass *parent* on to the function generated by Sun WorkShop Visual.

Note that the initialize function is called immediately after the create function and so you don't have to manage the widgets if you do it in the initialize function.

Initialize Function

The initialize function is called every time the user clicks on the resource button to pop up the dialog. The first time the dialog is invoked, the initialize function is called after the create function. The function should make the dialog visible by managing it and initialize any fields in it. The initialize function is passed the source text widget, the currently selected widget and the resource name.

```
void popup_initialize( Widget source_text, Widget current, char
*resource_name)
```

The *source_text* parameter is the source text widget on the resource panel. This widget can be used to obtain the resource value currently displayed on the resource panel. Since the source text widget is used to pass back the new value from the dialog, the initialize function should also save the source text widget in a static variable so that it is available later.

current represents the currently selected widget. The initialize function should check whether the currently selected widget has the expected resource because the user can invoke the popup dialog from the resource panel after selecting a widget of a different type. Note that *current* is *NULL* if the user has deleted all widgets in the hierarchy.

resource_name contains the resource name (a string such as "label"), not the defined name. This parameter is especially useful when you use the same popup to set more than one resource.

Update Function

The update function is called for every popup dialog each time the selection changes in the widget hierarchy.

```
void popup_update( Widget current )
```

current represents the newly selected widget. The update function should make the "Apply" button insensitive if the newly selected widget is of a different class, or if *current* is *NULL*. If you use the same dialog to set multiple resources, the safest approach is to make the "Apply" button insensitive in all cases. The user then has to click on the resource button again in order to use the popup. This extra step invokes the initialize function and ensures that the intended resource is set.

Popup Example

This example shows a simple resource popup consisting of a slider that is used to select an integer value. When the user clicks on "Apply" in the popup, the slider's value is converted to a text string and placed into the text widget in the resource panel.

The dialog itself was built in Sun WorkShop Visual. The Shell for the dialog was designated a Data Structure resulting in the structure shown below. Named widgets in the dialog are easy to access through the structure pointer *foo_dialog*. For example, *foo_dialog->scale* accesses the Scale widget.

```
typedef struct foo_dialog_s {
    Widget foo_dialog;
    Widget form;
    Widget scale;
    Widget apply;
    Widget close;
} foo_dialog_t, *foo_dialog_p;
static foo_dialog_p foo_dialog = (foo_dialog_p) NULL;
```

The following function is generated to create the Shell and all its children. The body of the generated function is omitted.

```
foo_dialog_p create_foo_dialog (Widget parent)
{
/* Sun WorkShop Visual generated code to create the dialog omitted here.*/
}
```

In the module prelude a static variable is created to hold the source text widget. The initialize function provides the source text widget.

```
static Widget source_text;
```

The dialog has an “Apply” button and a “Close” button, each with an Activate callback. The “Apply” button invokes the callback function shown below. This callback function gets the current value of the Scale, converts it to a text string and sets it into the source text widget. Note that this doesn’t set the resource; the user must still click on “Apply” on the resource panel.

```
static void
foo_do_apply (Widget w, XtPointer client_data, XtPointer call_data )
{
    int i;
    char buf[52];
    XmScaleGetValue ( foo_dialog->scale, &i );
    sprintf ( buf, “%d”, i );
    XmTextSetString ( source_text, buf );
}
```

The dialog also has a “Close” button. The Activate callback function on this button simply unmanages the child of the dialog’s Shell widget.

```
static void
foo_do_close (Widget w, XtPointer client_data, XtPointer call_data )
{
    XtUnmanageChild ( foo_dialog->form );
}
```

The create function calls the Sun WorkShop Visual generated creation function and saves the widget structure.

```
foo_create( Widget parent )
{
    foo_dialog = create_foo_dialog( parent );
}
```

The initialize function extracts the text from the source text field, converts it to an integer and sets the Scale to reflect the current value. It saves the source text field and so the new value set using the Scale can be applied to the source text field. It enables or disables the “Apply” button depending on the class of the current widget and makes the dialog visible.

```
foo_initialize( Widget text, Widget current)
{
    char *source_value;
    int i;
    source_text = text;
    source_value = XmTextGetString( source_text );
    i = atoi( source_value );
    XtFree( source_value );
    XmScaleSetValue( foo_dialog->scale, i );
    XtSetSensitive( foo_dialog->apply,
         current && XtIsSubclass (current, fooWidgetClass ) );
    XtManageChild( foo_dialog->form );
}
```

The update function enables or disables the “Apply” button, depending on the class of the current widget.

```
foo_update( Widget current )
{
    XtSetSensitive( foo_dialog->apply,
         current && XtIsSubclass( current, fooWidgetClass ) );
}
```

Resource Memory Management

Sun WorkShop Visual assumes a default memory management model for XmString and String (char *) type resources. For XmString type resources this model assumes that the widget will copy the XmString both on SetValues and on GetValues (i.e. the application can free the XmString after a GetValues or SetValues).

For String resources it is assumed that the widget copies the String on SetValues but not on GetValues (i.e. the application only frees a String after a SetValues). If you have a resource that does not conform to this model (typically an XmString resource that is not copied by the widget on GetValues), then you can override Sun WorkShop Visual's default behavior using the two Option Menus in the resource section. Where an XmString is not copied on GetValues, you should set the GetValues Option Menu to “Don’t Free” in the Widget Edit dialog. Similarly, if you have a String resource that is copied by the widget on GetValues (for example XmNmnemonicCharset in a Label Widget) then you should set the GetValues Option Menu to “Free”.

If you wish Sun WorkShop Visual always to default to not freeing the resources, set the “Default means Don’t Free” toggle.

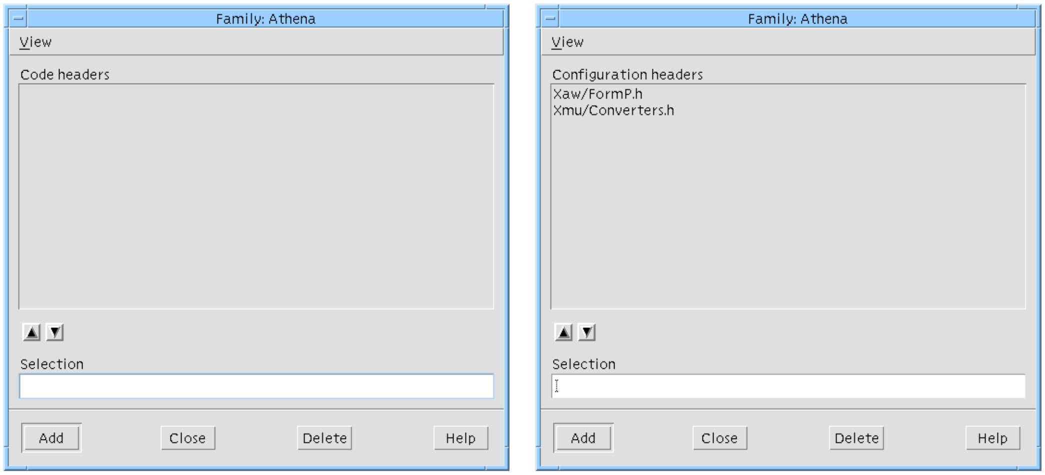
It is important to make sure that Sun WorkShop Visual does not free memory that it should not free as this will cause Sun WorkShop Visual to crash. It is less important if Sun WorkShop Visual is not freeing memory that it should free, as this will simply accumulate as a memory leak.

XmStringTable Resources

For Sun WorkShop Visual to handle XmStringTable resources correctly, you must also specify the integer type resource which is used as a count for the number of entries in the table. Add a resource specification for the XmStringTable.

Headers

visu_config generates two code modules, the Config file and the Code file. visu_config lets you specify a list of headers for each of these files. To specify these headers, use the Family Edit Dialog. Select “Code Integration Headers” from the View Menu to display the “Code headers” page and select “Config Integration Headers” to display the “Configuration headers” page. Both pages are shown in Figure 23-11.



Xaw/FormP.h
Xmu/Converters.h

FIGURE 23-11 The Code and Config Headers Pages of the Family Edit Dialog

There is a separate list of headers for each file. To add a header, type the filename, without quotes or angle brackets and click on “Add”. To delete a header, select it and click on “Delete”. To reorder a list, select any entry and use the arrow buttons to move it up or down.

The Code file defines the widget class records for user-defined widget classes. visu_config automatically generates a *#include* for the widget class header file which it takes from the Widget Edit dialog. Often no additional Code headers are needed.

The Config file contains a list of user-defined widgets, enumerations, and aliases. Widget headers for user-defined classes aren’t automatically generated to this file. If you configure visu_config with non-standard enumerations or resource types, include the header file in which they are defined.

The easiest way to find out what headers are needed is to generate and compile the code. If the compiler returns an undefined reference, find out which header contains the necessary definition and add it to the header list for that file. Then regenerate the code and try again.

Motif Widgets Stop List

You can use visu_config to *stop* selected Motif widgets from appearing in Sun WorkShop Visual. Stopped widgets do not appear in the widget palette. They work correctly if read in from an existing design file but cannot be selected in the hierarchy or created interactively. For example, you can use this feature to prevent users from using the PanedWindow widget if it isn’t in your company’s style guide.

To stop a widget, pull down the Edit Menu in the main visu_config dialog and select “Stop list” to display the dialog shown in Figure 23-12:

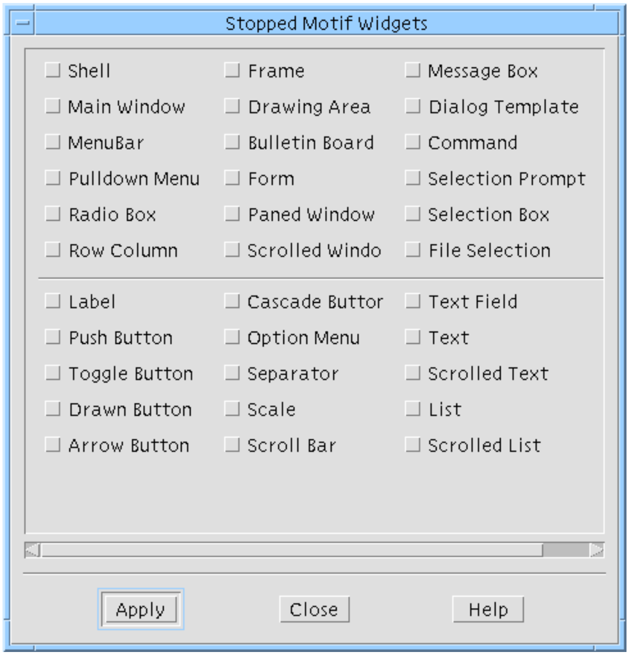


FIGURE 23-12 The Stopped Motif Widgets Dialogs

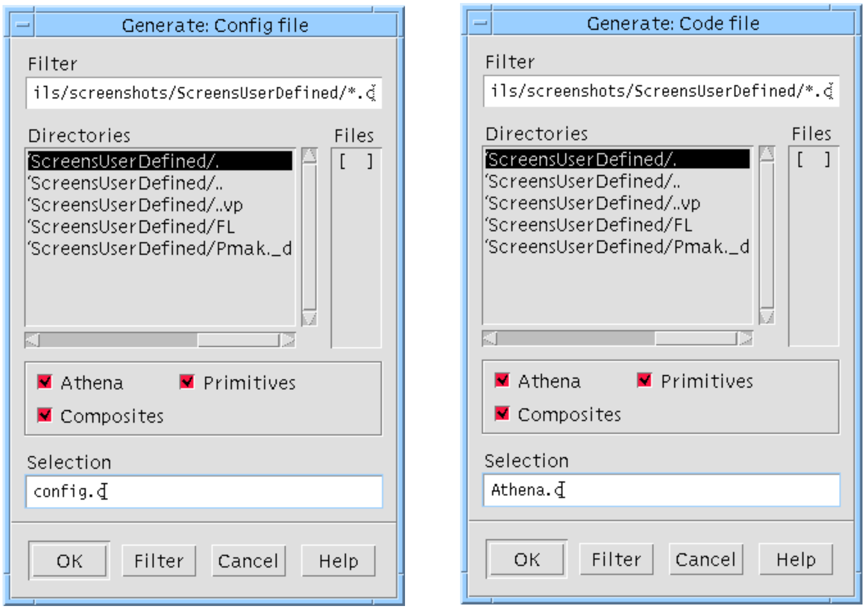
To remove a Motif widget from the widget palette, set the appropriate toggle and click on “Apply”.

Widgets can also be stopped by setting the visu.*stopList* resource, as described in “Configuration” on page 883. Note that widgets stopped using the resource can be reactivated easily using the resource file, while widgets stopped in visu_config can only be reactivated by rebuilding Sun WorkShop Visual.

User-defined widgets cannot be stopped using this dialog. You can select which user-defined families to make available in Sun WorkShop Visual via the visu_config Generate dialog, as discussed in the following section.

Generating and Compiling Code

The Generate Menu has two options, Config and Code which are used to generate the two configuration files. The pages displayed for each option are similar, as shown in Figure 23-13. Use the toggle buttons to select the families you want to include. Generate both files, using the same set of families for each.



Generate: Config file

Filter: ils/screenshots/ScreensUserDefined/*.c

Directories:
'ScreensUserDefined/.
'ScreensUserDefined/..
'ScreensUserDefined/..vp
'ScreensUserDefined/FL
'ScreensUserDefined/Pmak._d

Files: [ ]

☑ Athena ☑ Primitives
☑ Composites

Selection: config.c

Generate: Code file

Filter: ils/screenshots/ScreensUserDefined/*.c

Directories:
'ScreensUserDefined/.
'ScreensUserDefined/..
'ScreensUserDefined/..vp
'ScreensUserDefined/FL
'ScreensUserDefined/Pmak._d

Files: [ ]

☑ Athena ☑ Primitives
☑ Composites

Selection: Athena.c

FIGURE 23-13 The Config and Code Generate Dialogs

Compiling

The example makefile in $VISUROOT*/user_widgets/Athena*[1] compiles a Config file named *config.c*, a Code file named *Athena.c* and a file containing configuration functions, *Athenaextras.c*. These files and all the Athena icon bitmaps, are located in $VISUROOT*/user_widgets/Athena*. You can use this makefile to compile the Athena example. The result is an executable file called `visu`*.bin*.

When you configure with other widgets, use the example makefile as a starting point. Make sure that the makefile references all directories containing icon bitmap files and required header files.

If the compilation fails, inspect the generated code to find the problem. The cause may be a missing header. You can supply additional headers on the Code and Config Headers pages of the Family Edit dialog. The compiler also catches any misspellings in any of the visu_config dialogs. Fix the problem, regenerate the Code and Config files, then recompile.

Note that the linker detects any misspelled function names. If the misspelling occurred in visu_config, correct the problem, regenerate the Code and Config files, then recompile.

[1] $VISUROOT is the path to the Sun WorkShop Visual installation root directory.

Using the Widgets in Sun WorkShop Visual

The executable file is invoked with the correct environment by setting the environment variable USER_WIDGETS to the name of the user widgets directory (in this example “Athena”) and invoking the standard `visu` command in $VISUROOT/bin. This causes the local bitmaps or color_icons directories to be added to $XBMLANGPATH and the local app-defaults directory to replace $VISUROOT/lib/locale/<LANG>/app-defaults in $XFILESEARCHPATH.

A site-wide default can be set up by making a symbolic link called “local” in $VISUROOT/user_widgets to the user widgets directory of choice. In this case all users will get that version of Sun WorkShop Visual by default unless they explicitly override it with a setting of USER_WIDGETS. When USER_WIDGETS contains a value that is not recognized, or the `visu`.bin in that user widgets directory has not been built, the site default, if configured, or original “vanilla” Sun WorkShop Visual is invoked instead.

Accessing Abstract Children

Sun WorkShop Visual allows you full access to the abstract children of third party composite widgets. The Motif ScrolledWindow is an example of a composite widget - the scrollbars are the abstract children.

The following resource allows control over the accessibility of abstract children:

```
visu.abstractObjects: true
```

The resource defaults to “true”, which allows access to the children of third party widgets. Setting this to “false” will result in no children being displayed.

You can configure the accessibility of composite widget in visu_config, see “Can Edit as Abstract Child” on page 645.

Children of third party widgets can be fully configured through their resource panels. They cannot, however, be cut from your designs.

In the generated code, Sun WorkShop Visual calls `XtNameToWidget` in order to gain access to a component of a third party widget. Such components will normally have unique names. If, however, the widget names are not unique, using `XtNameToWidget` will not work correctly. In such a case, you will have to edit the generated code and use another means of accessing the widget.

Sun WorkShop Visual does not generate cross-platform code (MFC or Java) for the children of third party widgets. A place holder, such as a Canvas, is added in place of the root widget.

Testing the Configuration

This section describes a recommended testing procedure for the Sun WorkShop Visual interface for a user-defined widget. Use visu_config as suggested to fix any problems. Note that these tests are designed to detect problems with the way in which the widget was configured into Sun WorkShop Visual; they do not test the widget itself.

Creating a Widget

Note – This test is not appropriate if you turned off the “Can create widgets” toggle in the widget attributes panel.

Run Sun WorkShop Visual and verify that the icon for the user-defined widget is correct. If you use the small screen Sun WorkShop Visual, invoke Sun WorkShop Visual with the name `small_visu` and verify that the icon is correct in this case, too.

Create a hierarchy that contains an instance of your widget. If Sun WorkShop Visual fails when the widget is added, you may need a Realize function. For details, see “Configuration Functions” on page 672. If the failure is accompanied by an X error message about zero height/width windows, try using *sizedCreate()* (found in *Athenaextras.c*) as the Realize function for the widget.

If this does not work, try setting the “Disable foreground swapping” toggle in visu_config.

Foreground Swapping

If you have not disabled foreground swapping, create a hierarchy containing the user-defined widget. Select the Shell in the hierarchy and then select the user-defined widget. Verify that the widget in the dynamic display highlights correctly when selected. If this causes a problem, set the “Disable foreground swapping” toggle in visu_config.

Defined Name

Create a dialog containing an instance of the user-defined widget with every resource set. Generate C from Sun WorkShop Visual and compile it. If it fails to compile, you may get a message like this:

```
XtNfoo undefined
```

If you get such a message, first verify that the generated code includes the right public header for the widget class. If it doesn't, correct the header in the "Include file" field of visu_config's Widget Edit dialog.

If the header is being generated correctly but you still have compilation problems, you may need a Defined Name function. Look in the public header for a line such as:

```
#define <something> "foo"
```

If <*something*> is not *XtNfoo*, you need a Defined Name function. For details, see "Configuration Functions" on page 672.

Pages

If you have specified that resources for the user-defined widget should appear on specific pages, verify that they do and that all the required pages are present.

Converters

Display the "Miscellaneous" page of the resource panel for the user-defined widget. Verify that you can type valid resource values into the text widgets and that they are correctly applied to the widget. If you get a message indicating that there is no resource converter, you need to use the "Add in Sun WorkShop Visual" setting in visu_config.

Enumerations

Display the "Settings" page of the resource panel for the user-defined widget. Make sure the option menus all have the default value in parentheses at the top of their menus.

Display the "Miscellaneous" page if there is one. Enumeration resources appear on this page if they weren't configured in visu_config. If enumeration resources do appear on this page, go back and add them using visu_config.

Set each value for the enumeration in turn, including the default. Verify that each value works as expected in the dynamic display and that the generated code compiles correctly.

Popup Dialogs

If you specified custom popup dialogs for any resources, display the page of the resource panel on which each resource appears. Verify that the resource panel displays a button for each resource with a popup dialog. Click on the button. Verify that the dialog appears and is correctly initialized with the current value for that resource.

Set the resource in your dialog. Verify that the text widget on the resource panel updates correctly. Apply the setting from the resource panel and verify the result in the dynamic display.

If your dialog has a "Close" button, verify that it works as expected and that the dialog reappears when you click on the button on the resource panel.

Code Inspection

Finally, verify that the generated code is correct. To check the generated code, set each resource in turn, generate a C code file and an X resource file and inspect them to see that you get what you expect.

Code inspection for all the Motif widgets forms part of the Sun WorkShop Visual release process. Therefore if you have a user-defined widget that is derived from a Motif widget, you can concentrate on testing resources that are specific to the user-defined widget. This is also true if you have a user-defined widget that is derived from another user-defined widget that you have already tested.

Configuration Functions

visu_config lets you provide configuration functions to customize Sun WorkShop Visual's handling of user-defined widgets. This section provides definitions and examples of the configuration functions.

To add a configuration function, specify the name of the function on visu_config's Widget Edit dialog for the widget class, then regenerate the Code and Config files from visu_config. Edit your makefile to compile and link the file containing the code for your configuration functions.

For examples of the configuration functions that were used to integrate the Athena widgets into Sun WorkShop Visual, see:

`$VISUROOT/user_widgets/Athenaextras.c`[1].

Realize Function

By default, Sun WorkShop Visual creates widgets in the dynamic display by calling *XtCreateWidget()*. You can supply a Realize function to substitute for this. A Realize function is only needed for widgets that cause problems when created in Sun WorkShop Visual.

Realize Function Prototype

The Realize function has the following form. Note that it takes the same parameters and returns the same result as *XtCreateWidget()*.

```
Widget realize( char *name, WidgetClass class, Widget parent,
ArgList args, Cardinal arg_count )
```

The *ArgList* passed to a Realize function is always empty.

Realize Function Example

Some composite widgets, such as the Athena Form widget, cannot be realized without children unless their dimensions are explicitly set at creation time. Otherwise the widget is created at zero size, causing an X error. To solve this problem in the dynamic display, you can supply a Realize function like the one shown below, found in *Athenaextras.c*. This function initializes the widget's width and height resources to non-zero values, then calls *XtCreateWidget()* and returns the result.

```
WidgetsizedCreate( char *name, WidgetClass class, Widget parent,
ArgList args, Cardinal arg_count )
{
    Arg al[2];
    int ac=0;
```

[1] $VISUROOT is the path to the Sun WorkShop Visual installation root directory.

```
    XtSetArg(al[ac], XtNheight, 20); ac++;
    XtSetArg(al[ac], XtNwidth, 20); ac++;
    return XtCreateWidget ( name, class, parent, al, ac);
}
```

The Realize function is only used when Sun WorkShop Visual creates the widget in the dynamic display. It has no effect in the generated code.

Defined Name Function

In order to generate both code files and X resource files, Sun WorkShop Visual uses both the resource name, such as *label*, and the corresponding defined name, such as *XtNlabel*. Sun WorkShop Visual gets the name directly from the widget class record. By default, Sun WorkShop Visual derives the symbolic constant from the name by adding an *XtN* prefix.

If your widget doesn't follow this convention, you can configure Sun WorkShop Visual with a Defined Name function. The Defined Name function is a custom procedure that converts a resource name to its corresponding symbolic constant. To find out whether a widget needs a Defined Name function, look in the public header for the widget class. The header file contains lines like the following that define the symbolic constant and its value:

```
#define XtNlabel "label"
#define XtNfont "font"
#define XtNinternalWidth "internalWidth"
```

You need a Defined Name function if any of the defined names don't follow the naming convention. For example, many widget toolkits, including Motif, use a different prefix:

```
#define XmNbuttons "buttons"
#define XmNbuttonSet "buttonSet"
#define XmNbuttonType "buttonType"
```

Defined Name Function Prototype

The Defined Name function has the following form:

```
char *defined_name ( char *name )
```

The Defined Name function is passed a character string containing a resource name and should return a character string containing the corresponding defined name. Your Defined Name function can refer to Sun WorkShop Visual's internal Defined Name function, *def_defined_name()*. This function simply adds the default *XtN* prefix to the resource name.

Note – The function names *`defined_name`* and *`def_defined_name`* are already used by Sun WorkShop Visual so you should make sure that you do not use these names.

Defined Name Function Example

The defined name for the Athena Clock widget resource *hands* is not *XtNhands* but *XtNhand*. Therefore, the Clock widget needs the following Defined Name function:

```
char *clock_defined_name( char *name )
{
     /*
      * XtNhand is defined as hands, so can't just put
     *     XtN on the front
      */
     if ( strcmp ( name, "hands" ) == 0 )
           return "XtNhand";
     return def_defined_name ( name );
}
```

All Clock resources except *hands* follow the naming convention and so *def_defined_name()* is used to convert them.

Can Add Child and Appropriate Parent Functions

You can supply Appropriate Parent and Can Add Child functions to define the rules for valid parent-child relationships involving user-defined widgets. These rules control Sun WorkShop Visual features such as graying-out of palette icons, automatic selection of newly created widgets and dragging icons in the construction area.

Often these functions aren't needed. If you don't supply them, Sun WorkShop Visual uses the rules for the first known ancestor of the widget class. For example, if a user-defined widget is derived from the Primitive class, Sun WorkShop Visual uses the rules for the Primitive class and doesn't let the user add children to the widget.

The Appropriate Parent and Can Add Child functions are combined with rules for other widgets. For example, Sun WorkShop Visual already has a rule that MenuBar widgets can only have CascadeButtons as children, so you don't need an Appropriate Parent function to prevent the user from making your widget a child of a MenuBar. You only need to supply functions if your widget class has additional rules.

If you configure Sun WorkShop Visual with a non-Motif composite widget, the widget class should have a Can Add Child function to prevent it from having Motif children. Motif widgets assume that their parents are Motif widgets and Sun WorkShop Visual may core dump if a Motif widget is made a child of a non-Motif widget.

Appropriate Parent Function Prototype

The Appropriate Parent function is called when the user tries to add a widget of the user-defined class to the hierarchy, or tries to drag or copy the user-defined widget to another parent. This function determines whether the user-defined widget can be added as a child of the selected widget.

The Appropriate Parent function has the following form:

```
Boolean
is_appropriate_parent ( parent, childclass )
Widget parent;
WidgetClass childclass;
```

The first parameter is an instance of a widget in the hierarchy that is a proposed parent widget; the second is a pointer to your new widget class. The function should return *TRUE* if it is valid to add a child of your new class to the proposed parent widget and *FALSE* otherwise.

Because the Appropriate Parent function is passed the instance of the proposed parent widget, you can make rules based on either the class or the state of the parent widget. For example, you can check the parent widget's dimensions, ancestor widgets, or other children before letting the user add a new child of your user-defined class.

Appropriate Parent Function Example

By default, Sun WorkShop Visual lets Motif Manager widgets, such as the Form and Row Column, have children of any type. Appropriate Parent functions let you restrict the widget to being a child of only certain classes. For example, if your widget can only be a child of a DrawingArea, supply an Appropriate Parent function that returns *TRUE* if the proposed parent widget is a DrawingArea and *FALSE* if not. The code for this case is very simple:

```
Boolean
drawing_area_parent ( w, class )
Widget w;
WidgetClass class;
{
    if ( XtClass ( w ) == xmDrawingAreaWidgetClass )
         return True;
    return False;
}
```

Can Add Child Function Prototype

The Can Add Child function has the following form:

```
Boolean
can_add_child ( parent, childclass)
XWidget_p parent;
WidgetClass childclass;
{
...
}
```

This function is used for two purposes. Sun WorkShop Visual calls the Can Add Child function to determine whether a widget of a specific class is a valid child of the user-defined widget and calls the Can Add Child function to determine whether it should automatically select a newly created instance of the user-defined widget.

The first parameter is a pointer to an existing instance of the user-defined widget class. The parent widget instance is passed as an *XWidget_s* structure, an internal Sun WorkShop Visual data type that represents a widget instance. One field of this structure is a pointer to the widget instance. For documentation on the *XWidget_s* structure, see $VISUROOT*/user_widgets/hdrs/xwidget.h*.

The second parameter may be a pointer to a proposed child widget class, or may be *NULL*. If the second parameter is non-*NULL*, Sun WorkShop Visual is inquiring about a proposed child class. The function should return *TRUE* if the child can be added and *FALSE* if not. Note that you have a pointer to the parent *instance* but not to the child *class* because the child widget hasn't been instantiated. Your function may make rules based on the current state of the instance of the user-defined widget. For example, you can write a function that lets your widget accept only a limited number of children. If the second parameter is *NULL*, the user has just created a widget of this class.

If the Can Add Child function returns *TRUE*, Sun WorkShop Visual selects the newly created widget in the hierarchy; otherwise, the parent widget remains selected. In this case, the Can Add Child function should usually return *TRUE* if the widget can have children of any type and *FALSE* otherwise.

Can Add Child Example

The following example shows a Can Add Child function.

```
Boolean paned_can_add_child ( XWidget_p xw, WidgetClass class ) {

/* For newly created instance of this widget class, make the newly created widget
the currently selected widget in the hierarchy. */

if (class == NULL)
     return TRUE;

/* Allow all children except Drawing Area and ScrollBar. */

if ( class == xmDrawingAreaWidgetClass ||
     class == xmScrollBarWidgetClass )
     return False;
else
     return True;
}
```

Generating UIL

You can generate UIL for third party widgets. Any information Sun WorkShop Visual needs to do this is provided as part of the widget integration set[1]. If, however, you are integrating a new set of third party widgets, you will have to give Sun WorkShop Visual the extra information it needs in resources, as described below.

Resources for Third Party Widget UIL Code Generation

In order to generate UIL code for third party widgets, you have to provide Sun WorkShop Visual with enough information about the header files, creation procedures and template files. Most commonly used widget sets have already had this done for them - contact your Sun WorkShop Visual supplier for more details.

The following information is required when you wish to use a lesser known widget set:

1. UIL Header Files

2. UIL Creation Procedures

UIL Header Files

If the third party widget has a UIL header file associated with it, this is specified as follows:

```
visu.xw_<third_party_widget_class>.uilHeaderFile: foo.uil
```

For example, the XRT 3D widget is defined in this way:

```
visu*xw_XtXrt3d.uilHeaderFile: Xrt3d.uil
```

Note – UIL header files are supplied as part of the XRT widget set.

Many widget sets that do not come supplied with UIL header files, have had the header file created for them in order to enable UIL code generation from Sun WorkShop Visual. These can be found in:

```
$VISUROOT/user_widgets/USER_WIDGET_NAME/code_templates/UIL
```

where VISUROOT is the install directory of your Sun WorkShop Visual.

[1] Contact your Sun WorkShop Visual supplier for details of available widget integration sets.

The make templates generated for third party widgets have UILFLAGS set to include this code_templates directory automatically, as well as any normal widget-vendor locations. So for XRT the Makefile would contain the line:

```
UILFLAGS=-I${XRTHOME}/include/Xm ...
```

In this way, any third party widget with a UIL header will build.

UIL Creation Procedures

Non-motif widgets need to pre-declare creation procedures in any UIL header file, but not all suppliers do this. For this reason, there are two resources to control the required creator procedure that Sun WorkShop Visual must generate to get UIL to create the third party widget. Here are the three cases you must consider to work out which resource to use:

1. The supplied header file already has a procedure declaration.
2. The widget creator name has not been put in.
3. No pre-defined creator exists.
4. Special processing is required in order to create the widget.

Note – We cannot simply generate a declaration whatever the circumstance because UIL does not like multiple declarations of the same procedure. Hence we have two resources, so that Sun WorkShop Visual can tell when to generate the declaration and when not to.

These cases are described separately below.

A Procedure Declaration Exists

If there is already a procedure declaration in the header file, specify the name that Sun WorkShop Visual must generate like this:

```
visu*xw_<third_party_widget_class>.uilBuiltinProcedureName:
the_procedure_name
```

For example, XRT have the UIL creator XtCreateXrt3d already declared in Xrt3d.uil, so you would only need to tell Sun WorkShop Visual the name to generate:

```
visu*xw_XtXrt3d.uilBuiltinProcedureName: XtCreateXrt3d
```

Code like the following then appears in our generated UIL:

```
object some_third_party_object: user_defined
          the_object_creator_procedure_name
```

So for the sample XRT, Sun WorkShop Visual would generate something like:

```
include_file “Xrt3d.uil”;
object my_xrt_3d_variable: user_defined XtCreateXrt3d;
```

No Widget Creator Name Added

If the widget author has not put a widget creator name in, we need Sun WorkShop Visual to generate not only the call to some creator, but also a declaration.

If you use the following:

```
visu*xw_<third_party_widget_class>.uilProcedureName:
the_procedure_name
```

Sun WorkShop Visual would then generate the following:

```
include_file “AnySpecifiedUilHeader.uil”;

/* THIS NEXT LINE IS BAD IF AnySpecifiedUilHeader.uil already has this
*/
procedure the_procedure_name();
...
object my_third_party_variable: user_defined
                 the_procedure_name;
```

There Is No Pre-Defined Creator

Having no pre-defined creator means, for example, that there is no XmCreatePushButton to replace the lower level generic XtCreateWidget() call.

Sun WorkShop Visual will in this case generate:

1. A creator with the name constructed from the widget class into the UIL file.
2. A declaration for this into the UIL file.
3. An actual function with the same name in the C for UIL file to wrap this up.

The form of the generated constructor is XdUilCreate<widget_class_name>

This will automatically happen if Sun WorkShop Visual comes across some third party widget where no resources for UIL have been set.

For example, if we had a widget class `FredWidgetClass`, the UIL file would contain the following:

```
procedure XdUilCreateFredWidgetClass();
```

```
...
object fred_variable : user_defined XdUilCreateFredWidgetClass
```

And the C for UIL would contain:

```
...
Widget XdUilCreateFredWidgetClass(parent, name, argv, argc)
Widget   parent ;
String   name ;
Arg     *argv ;
Cardinal argc ;
{
     return XtCreateWidget(name, fredWidgetClass, parent, argv,
           argc) ;
}
```

Special Processing Needed for Widget Creation

If special processing is required to create one of the third party widgets but no UIL creator procedure has been declared, a procedure is needed in standard form with the contents pre-configured.

This is done through the following resource:

```
visu*xw_<widget_class>.uilProcedureTemplate: some_file_name
```

Then, `some_file_name` is simply written out, as is, between the curly brackets of any UIL procedure Sun WorkShop Visual has to generate into the C for UIL.

For example, if the fred widget had to have its size set *before* being created, we would have to:

1. Create a file called `fred.uil_template`
2. Put this into $USER_WIDGETS/code_templates/UIL
3. Set the resource:

```
XDesigner*xw_fred.uilProcedureTemplate:
        $USER_WIDGETS/code_templates/UIL/fred.uil_template
```

Note – The value of this resource can take shell variables that are then expanded automatically.

The next step is to write this template file `fred.uil_template` as follows:

```
/* START OF UIL TEMPLATE FILE */
Arg     *av ;
Cardinal ac ;
Widget   w ;
/*  Copy down any passed arguments */
     av = (Arg *) XtMalloc((unsigned) (argc + 2) * sizeof(Arg)) ;
     for (ac = 0 ; ac < argc ; ac++) {
           av[ac].name  = argv[ac].name ;
           av[ac].value = argv[ac].value ;
     }
/* Add the necessary width, height */
     XtSetArg(av[ac], XmNwidth,  100) ; ac++ ;
     XtSetArg(av[ac], XmNheight, 100) ; ac++ ;
     w = XtCreateWidget(name, fredWidgetClass, parent, av,
                                                  ac) ;
/* tidy up */
     XtFree((char *) av) ;
     return w ;
/* END OF UIL TEMPLATE FILE */
```

X-Designer here will go through its normal algorithms for deciding the actual name of the UIL procedure required, then will put the above between the curly brackets of this UIL procedure in the C for UIL file.

CHAPTER 24

Command Line Operations

Introduction

This chapter describes the command line switches understood by Sun WorkShop Visual. They fall into two categories: those which affect Sun WorkShop Visual running interactively and those which can be used for command line code generation. This chapter also describes the command line versions of Sun WorkShop Visual Capture and Sun WorkShop Visual Replay and the commands provided for conversion of UIL and GIL code into Sun WorkShop Visual save files.

Command Line Switches for Interactive Use

The following command line switches are available:

TABLE 24-1 `visu` Command Line Options for Interactive Use

Switch	Meaning
windows	Start Sun WorkShop Visual in Microsoft Windows mode
f file	Specify input file

TABLE 24-1 `visu` Command Line Options for Interactive Use *(Continued)*

Switch	Meaning
L	Use private colormap
x	Display this explanation (and exit)
V	Display Sun WorkShop Visual version information (the program is not run)

For further information on starting Sun WorkShop Visual in Microsoft Windows mode see "Starting in Microsoft Windows Mode" on page 360.

Using a private colormap is useful if you intend to use a lot of colors in the pixmap editor. See "Editing Pixmaps" on page 148 for a description of the pixmap editor. If you do select this option, however, you may observe strange color effects in other windows.

Generating Code From the Command Line

The command line synopsis is:

```
visu [-csepAKCSEulbarmRMFWX [code_file]] [-G directory [-O]]
[-windows] -f filename
```

TABLE 24-2 `visu` Command Line Options

Switch	Code file generated
c	C
s	C stubs
e	C externs
p	C/C++ pixmaps
A	Force ANSI C (use with **-c**, **-s** and **-e**)
K	Force K&R C (use with **-c**, **-s** and **-e**)
C	C++
S	C++ stubs
E	C++ externs
u	UIL

TABLE 24-2 `visu` Command Line Options *(Continued)*

Switch	Code file generated
l	C for UIL
b	C externs for UIL
a	UIL pixmaps
r	X resource file
m	Makefile
X	X resource file (synonym for r)
M	Generate Motif flavor C++
F	Generate Motif XP flavor C++
J	Generate Java to the directory specified as code_file
-G directory	Generate to this directory (as if from the Generate Dialog)
-O (with -G)	Objects only. Generate no main code
W	Generate Microsoft Windows MFC flavor C++
pixmaps	Generate all pixmaps to separate .xpm files in the directory specified as code_file
R	Microsoft Windows resource file
windows	Start Sun WorkShop Visual in Microsoft Windows mode
f file	Specify input file

code_file represents the file to be generated. If you do not specify a *code_file*, Sun WorkShop Visual generates code to the last target file specified in your source file for the given language.

For Java code generation (-J) and pixmaps (-pixmaps) multiple source files may be generated and the names of the code files are the names of the classes (for Java) or the pixmaps used in your design. Therefore, *code_file* is the target directory for these source files.

filename represents the design file (*.xd*) to be used as a source for the code generation. You must always specify a *filename*. If you do not also specify a *code_file*, use the *-f* separator to indicate that you are providing only one filename.

The -*windows* switch specifies Microsoft Windows mode. For further information on starting Sun WorkShop Visual in Microsoft Windows mode see “Starting in Microsoft Windows Mode” on page 360.

The M, *F*, *W* and *R* switches are only used in conjunction with the -*windows* switch.

Examples

The command:

```
visu -c foo.c -f foo.xd
```

generates C code from the design in *foo.xd* into the file *foo.c*.

The command:

```
visu -c -f foo.xd
```

generates C code from *foo.xd* into the target file that was specified the last time C code was generated from *foo.xd* via the Generate Dialog.

You can use a single command to generate multiple files using one of the following forms:

```
visu -c -e -s -f foo.xd
```

or

```
visu -c <c_file> -e <extern_file> -s <stub_file> -f foo.xd
```

Sun WorkShop Visual exits with status zero if successful and non-zero status if it fails to generate the code for any reason.

Trouble-Shooting

Sun WorkShop Visual must be connected to an X server to generate code from the command line. Usually command line code generation does not create any visible windows but windows do appear momentarily on the server screen for designs containing certain types of widgets, such as ScrolledList and ScrolledText and when generating Microsoft Windows code.

If you don't specify a *code_file*, Sun WorkShop Visual relies on the filename saved in the design file for the specified type of code. The filename is only saved when you specify it on the Generate Dialog and then save the file. If you have never used the Generate Dialog to generate this type of code from the design file, Sun WorkShop Visual produces only an error message.

In all cases, the generate toggles are set as they were last saved in the design file. If you have never generated this type of code from the design file, default toggle settings are used.

Sun WorkShop Visual Replay

Sun WorkShop Visual Replay (when used to record user actions) is supplied as a stand-alone application called visu_record. .

The following line shows how to use visu_record:

```
visu_record -f MyRecordScript AnApplication
```

`MyRecordScript` is the name of a file into which a script recording the session will be saved. `AnApplication` is the name of the application you wish to record.

The following line shows how to use `visu_replay`:

You can leave out the filename argument for both Sun WorkShop Visual Replay and XD/Record. XD/Record will then output to standard output and Sun WorkShop Visual Replay will read from standard input.

The following table shows the full list of command line switches available for both visu_record and `visu_replay`:

TABLE 24-3 visu_record/`visu_replay` Command Line Options

Switch	Meaning
x	Display information about the tool
f testscript	Save to file (otherwise stdout) for visu_record Read from file (otherwise stdin) for `visu_replay`
use n	Skip n shells before the real Application Shell
lang locale	Run visu_record/`visu_replay` (including the graphical user interface and all error messages) in locale and ignore any LANG settings
p	Preprocess script with C preprocessor before replaying
t target	Use ’target’ alternative internal libraries if available
v	Verbose output
V	Print visu_record/`visu_replay` version information
w	Print summary information about the display, server and window manager
O	Override (program exit for non-motif applications
i	interactive (uses the Capture/Replay dialog - ignores ‘-f’)
I	Force the dialog to appear

TABLE 24-3 visu_record / visu_replay Command Line Options *(Continued)*

Switch	Meaning
exit-on-error	Terminate the visu_replay script and the application if a command cannot be replayed
user-on-error	Terminate the visu_replay script and the application if a command cannot be replayed but remain in the application
skip-on-error	Jump to the next sequence in the script if a visu_replay command cannot be replayed
interval ms	Allow ms (milliseconds) between execution of commands
ignore-server-time	Allow the client to run as fast as possible

Sun WorkShop Visual Capture

Sun WorkShop Visual Capture is supplied as a stand-alone application called visu_capture.

The following line gives an example of how visu_capture can be used:

```
visu_capture AnApplication
```

This displays the Capture dialog and runs the application, AnApplication.

The following table shows the full list of command line switches available:

TABLE 24-4 visu_capture Command Line Options

Switch	Meaning
x	Display information about visu_capture
f file	Save to file. No Capture dialog is displayed with this option
lang locale	Run visu_capture (including the graphical user interface and all error messages) in locale and ignore any LANG settings
t target	Use ’target’ alternative internal libraries if available
use n	Skip n shells before the real Application Shell
v	Verbose output
V	Print visu_capture version information

TABLE 24-4 `visu_capture` Command Line Options *(Continued)*

Switch	Meaning
w	Print summary information about the display, server and window manager
O	Override (program exit for non-motif applications
i	interactive (uses the Capture/Replay dialog - ignores '-f'). This is the default behavior
I	Force the dialog to appear
j	Java-Ready Capture
static-design	Take a single snapshot of the Application Shell

Converting UIL Source to Sun WorkShop Visual Save Files

The *uil2xd* filter converts UIL source code to Sun WorkShop Visual save files. It reads UIL source from standard input and writes a save file for Sun WorkShop Visual on standard output.

By default, *uil2xd* generates a save file for the latest release of Sun WorkShop Visual. The command line synopsis is:

```
uil2xd [-tlxywhpsaX] [-I include_dir]
```

The command line options are listed in the following table:

TABLE 24-5 uil2xd Command Line Options

Switch	Meaning
t	Don't convert ScrolledWindow containing Text to Scrolled Text. By default **uil2xd** converts a ScrolledWindow widget containing a Text widget into a ScrolledText widget. Use the -t flag to preserve the structure.
l	Don't convert ScrolledWindow containing List to Scrolled List. By default **uil2xd** converts a ScrolledWindow widget containing a List widget into a ScrolledList widget. Use the -l flag to preserve the structure.
x	Pass through XmNx resources. By default **uil2xd** does not output absolute positions in the save file. Use the -x flag to pass XmNx resources into the output file.
y	Pass through XmNy resources. By default **uil2xd** does not output absolute positions in the save file. Use the -y flag to pass XmNy resources into the output file.
w	Pass through XmNwidth resources. By default **uil2xd** does not output absolute sizes in the save file. Use the -w flag to pass XmNwidth resources into the output file.
h	Pass through XmNheight resources. By default **uil2xd** does not output absolute sizes in the save file. Use the -h flag to pass XmNheight resources into the output file.
p	Preserve position resources in output file. Same as -x -y
s	Preserve size resources in output file. Same as -w -h
a	Preserve position and size resources in output file. Same as -p -s
e	Explain how to recover from syntax errors
A	Generate fake attachments for unattached form children
I include_dir	Adds include_dir to the list of directories that will be searched for include files.
X	Print list of switches.

uil2xd does not handle the following constructs:

- String tables containing compound strings
- Color_table
- Icon
- Ascii tables in argument definition
- Integer tables in argument definition
- Imported keyword – this is a fatal error
- Exported keyword
- Private keyword
- Creation procedure
- Default character set clause
- Identifier section

Except for the imported keyword, *uil2xd* simply ignores these constructs.

Converting GIL Source to Sun WorkShop Visual Save Files

The *gil2xd* filter converts Sun Microsystems Inc.’s DevGuide save files into Sun WorkShop Visual save files. The converter works by mapping the OPEN LOOK objects into Motif objects. It reads GIL source from standard input and writes a save file for Sun WorkShop Visual on standard output.

It generates a save file for the latest version of Sun WorkShop Visual. The command line synopsis is:

```
gil2xd [-xywhpsaX]
```

The command line options are listed in the following table:

TABLE 24-6 gil2xd Command Line Options

Switch	Meaning
x	Pass through XmNx resources. By default **gil2xd** does not output absolute positions in the save file. Use the -x flag to pass XmNx resources into the output file.
y	Pass through XmNy resources. By default **gil2xd** does not output absolute positions in the save file. Use the -y flag to pass XmNy resources into the output file.
w	Pass through XmNwidth resources. By default **gil2xd** does not output absolute sizes in the save file. Use the -w flag to pass XmNwidth resources into the output file.
h	Pass through XmNheight resources. By default **gil2xd** does not output absolute sizes in the save file. Use the -h flag to pass XmNheight resources into the output file.
p	Preserve position resources in output file. Same as -x -y
s	Preserve position and size resources in output file. Same as -w -h
a	Preserve position and size resources in output file. Same as -p -s
X	Print list of switches

gil2xd does not handle connections other than function calls and the simple notify actions for buttons which can be mapped to links. Other connections are reported as warnings. *gil2xd* simply ignores these constructs.

Mappings

Few of the mappings from OPEN LOOK objects to Motif widgets are straightforward as they depend somewhat on their context. The fundamentals of the mappings are outlined below.

base-window

Maps to a DialogShell with a MainWindow child with a Form work area.

popup-window

Maps to a DialogShell with a Form child.

canvas-pane

Maps to a DrawingArea which will be a child of a ScrolledWindow if horizontal-scrollbar or vertical-scrollbar is true. An associated PopupMenu is created as a child of the DrawingArea.

control-area

Maps to a Form.

menu

Maps to a Menu. If the menu has a menu-title attribute, the first child widget will be a Label which shows the title, followed by a Separator. The menu items are mapped to additional children of the Menu. If the menu-type attribute is command, the widgets will be ToggleButtons; if they have an associated menu they will be CascadeButtons, otherwise they will be PushButtons. As Sun WorkShop Visual has no concept of shared menus, menus which are referenced from more than one place will map to copies of the Menu.

message

Maps to a Label.

button

Maps to a PushButton if it does not have a menu, otherwise it maps to a CascadeButton. This CascadeButton will be created in a MenuBar. CascadeButtons which have the same y co-ordinate will be created in the same MenuBar. The MenuBar will be created in an enclosing MainWindow if possible, otherwise it will be created at the appropriate location.

slider and gauge

Both map to a Scale. Separators will be added as children for tick marks and Labels may be added to show the min-value-string and max-value-string. The min-value and max-value map to the Scale's minimum and maximum fields respectively.

setting

Maps to an OptionMenu if setting-type is stack, otherwise maps to a RowColumn. The choices are mapped to PushButtons in an OptionMenu and ToggleButtons in a RowColumn. For exclusive and non-exclusive settings the ToggleButton is adjusted so that the indicator is not used (shadowThickness = 2, marginLeft = 0, indicatorOn = false).

text field

Maps to RowColumn with Label and Text widgets. Text will be ScrolledText if text-type is set to multiline.

list

Maps to a ScrolledList. If the list has a label attribute set, the ScrolledList is created as a child of a RowColumn with a Label child which displays the label. If the list has a title attribute, the ScrolledList is created as a child of a Frame with a Label to display this title.

drop-target

Maps to a Label.

stack

Maps to a Form which has each of the stack members as children. The children are attached to both sides of the Form.

group

Maps to a RowColumn which has each of the member widgets as children.

term-pane and text-pane

Both map to ScrolledText.

Attributes

Once the gil objects have been mapped to widgets the attributes must be mapped to appropriate widget resources. The following resources are always mapped:

TABLE 24-7 Resources That Are Always Mapped

gil	xd	Notes
x	XmNx	
y	XmNy	
width	XmNwidth	
height	XmNheight	
foreground-color	XmNforeground	
background-color	XmNbackground	
initial-state	XmNsensitive	inactive - sensitive = false invisible - managed = false

The width and height resources are only used if the -w or -h flags are set when `gil2xd` is run. The x and y resources will be output if the -x or -y flags are set. However, for widgets which are children of Forms the x and y co-ordinates will be used to calculate default Form attachments to preserve the approximate layout.

Note that many of the Motif manager widgets will ignore explicit x, y, width and height resources anyway. The `gil2xd` filter can be used without any of the runtime flags to produce an adequate layout which can be easily modified using Sun WorkShop Visual.

Other resources are mapped to the nearest possible resource.

TABLE 24-8 Resources That Are Mapped to the Nearest Possible Resource

gil	xd	Notes
columns	XmNcolumns	
constant-width	XmNrecomputeSize	
group-type	XmNorientation	Sets XmNnumColumns, XmNorientation and XmNpacking to reproduce a similar layout of group
icon-file	XmNiconPixmap	
icon-label	XmNiconName	
icon-mask	XmNiconMask	
initial-state	XmNinitialState	DialogShell only
initial-value	XmNvalue	
label	XmNlabelString	If matching label-type attribute is glyph then label is mapped to labelPixmap
label	XmNtitle	For shells
label	XmNtitleString	For gauge
label-type	XmNlabelType	
layout-type	XmNorientation	
max-value	XmNmaximum	
menu-type	XmNradioBehavior	If exclusive, XmNradioBehavior = true
min-value	XmNminimum	
multiple-selections	XmNselectionPolicy	If set, XmNselectionPolicy = MULTIPLE_SELECT
orientation	XmNorientation	
pinnable	XmNtearOffModel	If pinnable, XmNtearOffModel = TEAR_OFF_ENABLED
read-only	XmNeditable	
resizeable	XmNallowResize	

TABLE 24-8 Resources That Are Mapped to the Nearest Possible Resource *(Continued)*

gil	xd	Notes
rows	XmNnumColumns	Sets XmNnumColumns, XmNorientation and XmNpacking to reproduce a similar layout of settings
rows	XmNrows	For text widget
rows	XmNvisibleItemCount	For list widget
selection-required	XmNradioAlwaysOne	
show-border	XmNshadowThickness	If set sets XmNshadowThickness to 1 for forms which are not children of a Shell
show-value	XmNshowValue	
slider-width	XmNscaleWidth	Sets XmNscaleWidth or XmNscaleHeight depending on orientation
stored-length	XmNmaxLength	
text-initial-value	XmNvalue	
text-type	XmNeditMode	If multiline, XmNscrollVertical = false, rows maps to XmNrows
title	XmNlabelString	
value-length	XmNcolumns	

Actions which have a CallFunction function_type are mapped to callbacks where appropriate.

TABLE 24-9 Actions That Are Mapped to Callbacks

Action	Callback	Widget
Create	XmNcreateCallback	Any
Destroy	XmNdestroyCallback	Any
Notify	XmNactivateCallback	PushButton
select	XmNinputCallback	DrawingArea
adjust	XmNinputCallback	DrawingArea
DoubleClick	XmNinputCallback	DrawingArea
Repaint	XmNexposeCallback	DrawingArea

TABLE 24-9 Actions That Are Mapped to Callbacks *(Continued)*

Action	Callback	Widget
Resize	XmNresizeCallback	DrawingArea
Select	XmNvalueChangedCallback	Gauge
Adjust	XmNdragCallback	Gauge
Notify	XmNvalueChangedCallback	Gauge
Popup	XmNmapCallback	Menu
Popdown	XmNunmapCallback	Menu
Notify	XmNentryCallback	Menu
Notify	XmNvalueChangedCallback	ToggleButton
Unselect	XmNvalueChangedCallback	ToggleButton
Popup	XmNpopupCallback	Shell
Popdown	XmNpopdownCallback	Shell
Notify	XmNactivateCallback	Text
KeyPress	XmNvalueChangedCallback	Text
Notify	XmNentryCallback	RowColumn
Done	XmNunmapCallback	Form
Notify	XmNbrowseSelectionCallback	List

There are a number of other gil actions which are not detailed in this list. These are not supported by the filter as there is no appropriate Motif callback.

Notify actions for PushButtons which have a Show, Hide, Enable or Disable connection are mapped to the appropriate Sun WorkShop Visual link.

CHAPTER 25

Configuration

Introduction

There are several ways to customize Sun WorkShop Visual, using either the resource file or visu_config. This section explains the main features that can be customized via the resource file. These features are:

- Callback and Prelude editing
- Palette icons
- Palette contents and layout
- Toolbar
- Makefile templates

For further information on Sun WorkShop Visual resources, see Appendix D, “Application Defaults”, starting on page 867. For information on using visu_config, see Chapter 23, “User-Defined Widgets”, starting on page 627.

Another area of Sun WorkShop Visual which can be configured using resources is the dynamic display window. This window has its own application class name and, therefore, its own resource file. Details are provided in “Dynamic Display Window” on page 713.

Setting up Callback and Prelude Editing

There are two mechanisms for editing callbacks and preludes. The first uses the Sun Edit Services and the second simply invokes an editor of your choice in an “xterm” window. This section describes the application resources, applications and environment variables needed to configure the editing mechanism. See Appendix D, "Application Defaults" for details on using Sun WorkShop Visual's application resources. This first resource controls whether or not you wish the callback editing mechanism to be present:

callbackEditing

This resource controls whether callback editing is enabled; if it is set to false then the buttons relating to this feature do not appear.

If you wish to use callback and prelude editing without using the Sun WorkShop Edit Server, you will need to specify which editor to use.

editor

This resource specifies the location of the shell wrapper script which starts up the editor. By default it is set to:

```
$VISUROOT/lib/scripts/xd_edit
```

`xd_edit` is a text file containing a shell script which provides the startup command applicable to different editors. If the editor you wish to use is not listed in this file, the default is to use the behavior specified for the editor "vi". When an editor is invoked, an "xterm" is started and Sun WorkShop Visual tries to move to the appropriate line in the stubs or primary source file (according to whether you are editing callbacks or preludes respectively). This is not applicable to all editors. Motif-based editors, for example, do not need an "xterm" and some editors cannot move to a specified line on startup.

The file `xd_edit` contains some examples for different types of editor along with explanatory comments. You can add special information for your chosen editor by copying existing lines. Refer to the documentation for your UNIX shell for details of the shell scripting language.

The `xd_edit` shell script checks the `EDITOR` environment variable for the name of the editor you wish to use. If it is not set, the script checks the `VISUAL` environment variable. If that is also not set, the default is "vi". If you wish to use either of these variables, specify the name along with the full directory name if it is not in a directory listed by your `PATH` environment variable. Once the script has decided on an editor, it runs the editor in the terminal program defined by the environment variable `XD_TERM`. If this is not set, the default is to use the `xterm` program.

See your UNIX documentation for details on using environment variables.

Using the Editing Facility

See “Adding Callback Functionality” on page 226 for details on how to use the editing facility to edit callbacks and “Code Preludes” on page 242 for a description of prelude editing.

Palette Icons

Sun WorkShop Visual has an icon for each widget class. The icon is drawn on the palette buttons and *displayed* in the tree hierarchy. The icon can be either a full color XPM format pixmap or a monochrome bitmap. On start-up, Sun WorkShop Visual *searches* through the application resources to find a pixmap or bitmap file for each icon. If one cannot be found, a built-in bitmap is used.

Specifying the Icon File

Each Motif widget has an application resource that specifies its icon file. The application resource file is:

*$*VISUROOT*/lib/locale/<your-locale>/app-defaults/*visu

where VISUROOT *is your* Sun WorkShop Visual *install directory and your-locale is your local language locale name. The default locale is C. The LANG environment variable may tell you what your locale name is.*

The application resource file contains a complete list of resource names for the Motif widget icons. For example, the resource for the ArrowButton icon is:

```
visu.arrowButtonPixmap: arrow.xpm
```

Sun WorkShop Visual searches in the same way as *XmGetPixmap()* to find the bitmap file. This search path is complex; for details, refer to the Motif documentation. In practice, Sun WorkShop Visual places the default pixmap files in *$*VISUROOT*/bitmaps* and adds *$*VISUROOT*/bitmaps/%B* to the XBMLANGPATH environment variable. Individual users can provide their own local pixmaps by creating a file of the correct name, such as *arrow.xpm*, in a directory and adding that directory with the file matching string “*/%B*” to the SW_VISU_XBMLANGPATH environment variable. For example,

```
setenv SW_VISU_XBMLANGPATH /home/me/pix/%B
```

Default Pixmaps

Sun WorkShop Visual comes with two sets of icon pixmaps. The default set is located in *$VISUROOT/bitmaps*. These icons are drawn using minimal color and will work on either color or monochrome screens.

A set of color pixmaps is located in *$VISUROOT/color_icons*. To change to the color pixmaps, either set the environment variable SW_VISU_ICONS to *color_icons*, or add *$VISUROOT/color_icons/%B* to the SW_VISU_XBMLANGPATH environment variable. To revert to the default icons either unset SW_VISU_ICONS or set it to *bitmaps*.

Alternatively, an individual user can specify a different file name by setting the resource in a local copy of the Sun WorkShop Visual resource file:

```
visu.arrowButtonPixmap: my_arrow.xpm
```

or:

```
visu.arrowButtonPixmap: /home/me/visu_bitmaps/my_arrow.xpm
```

Pixmap Requirements

You can create your own color pixmaps for icons using XPM3 format. This format can be created using the Sun WorkShop Visual Pixmap editor (see “Editing Pixmaps” on page 148). Icon pixmaps can be any size; the palette and tree are displayed with a vertical spacing to accommodate the tallest icon. The Sun WorkShop Visual display looks best when all icon pixmaps are the same size. Default sizes are 32 by 32 for the large-screen icon pixmap and 20 by 20 for the small-screen version.

When printed from Sun WorkShop Visual, the icons are *dithered* to grey scales.

Transparent Area

The icon should contain an area of transparency. Sun WorkShop Visual uses this area to display highlighting and structure colors in the tree and the background color on the palette button. XPM supports transparency by means of the color name “none” (not the color object “none”). See “Transparent Colors” on page 160 for instructions on setting up transparent colors.

Icons for User-Defined Widgets

visu_config's Widget Edit dialog lets you specify icons for user-defined widgets. For more information, see "Widget Attributes" on page 640.

Icons for Palette Definitions

The Edit Definitions dialog lets you specify an icon for each palette definition and a file name to be used as a fallback if the resource is not set. The pixmap file is searched for in the same way as the default pixmaps. If Sun WorkShop Visual cannot find the pixmap it will use the default pixmap for the widget that is at the root of the definition. See "Editing the Definitions File" on page 269 for further details.

Palette Contents

You can stop certain Motif widget classes from appearing on the Sun WorkShop Visual palette using the *stopList* resource. In addition, you can use visu_config to stop user-defined widgets appearing on the palette. Because stopped widgets do not appear on the palette, they cannot be created interactively. They can be read in from a save file generated by Sun WorkShop Visual but are not selectable.

To stop a Motif widget class, specify the class name in the *stopList* resource. For example, to remove the Motif PanedWindow and ArrowButton from your widget palette, set the resource:

```
visu.stopList: XmPanedWindow,XmArrowButton
```

To stop a user-defined widget, specify the class name:

```
visu.stopList: boxWidgetClass,formWidgetClass
```

visu_config has a Stopped Motif Widgets dialog which contains a toggle for each of the Motif widgets. For more information, see Chapter 23, "User-Defined Widgets", starting on page 627. Widgets that are removed from the palette in visu_config cannot be added back in using Sun WorkShop Visual resources.

Palette Layout

The default settings display a vertical palette containing three columns of widget icons and attached to the main window. You can change the default layout using the resource file. You can also change the palette layout at run time using the Palette Menu.

Separate Palette

You can display the palette in a separate window. To separate the palette at run time, use the “Separate Palette” option in the Palette Menu. To have a separate palette by default, set the following resource:

```
visu*pm_separate.set:true
```

When you separate the palette in the resource file, you must also explicitly set the default height of the Sun WorkShop Visual main window:

```
visu.main_window.height: 600
```

The resource file contains several examples of alternative layouts, such as:

! Two columns is good if you do not have user defined widgets

```
visu*icon_panel.composite_icons.numColumns: 2
visu*icon_panel.basic_icons.numColumns: 2
```

! Set both labels and icons on

```
visu*pm_both.set: true
```

! You might also want to make the tool a bit wider

```
visu.main_window.width: 750
```

! Four columns, with labels underneath

```
XDesigner*widgetPalette.composite.buttonBox.XmRowColumn.\
orientation: VERTICAL
XDesigner*widgetPalette.composite.buttonBox.numColumns: 4
XDesigner*widgetPalette.basic.buttonBox.XmRowColumn.\
orientation: VERTICAL
XDesigner*widgetPalette.basic.buttonBox.numColumns: 4
XDesigner*widgetPalette*xwidget_icons*orientation: VERTICAL
XDesigner*widgetPalette*xwidget_icons.numColumns: 4
XDesigner*widgetPalette*_defn_icons*orientation: VERTICAL
```

```
XDesigner*widgetPalette*_defn_icons.numColumns: 4
```

Toolbar

The Sun WorkShop Visual interface includes a toolbar. Buttons in the toolbar correspond to buttons in the menus. Generally, when you select a toolbar button, it does exactly the same thing as the corresponding menu button.

Configuring the Toolbar

To configure buttons into the toolbar, use the *toolbarDescription* resource. This should contain a comma-separated list of the widget names of the buttons. It can also contain the word *separator* to add extra space between items and the word *flavor* to insert the Microsoft Windows flavor option menu.

To obtain the widget name of a button, search Sun WorkShop Visual's application resource file for the entry that sets the *labelString* resource on the button in the menu bar. See Appendix D, "Application Defaults", for details on the location of application resource files.

Sun WorkShop Visual's resource file contains this line:

```
visu*em_cut.labelString: Cut
```

em_cut is the widget name of the "Cut" button in the Edit Menu.

The following line produces a toolbar with "Cut", "Copy", "Paste", "Core resources...", "Layout..." (layout editor), and "C" (code generation) buttons:

```
visu.toolbarDescription:separator,em_cut,\
em_copy,em_paste,separator,wm_prim,\
wm_layout,separator,gm_c
```

Labels for Toolbar Buttons

The toolbar buttons have the same widget name as the counterpart menu button and so, by default (assuming that no pixmaps are configured), they appear with the same words. For example, the resource file contains the line:

```
visu*gm_c.labelString: C...
```

By default, this applies to both the menu and the toolbar. You might want to remove the ellipsis for the toolbar version since code generation buttons in the toolbar do not display a dialog. To do this, add the following line:

```
visu*toolbar.gm_c.labelString: C
```

Pixmaps for Toolbar Buttons

The toolbar buttons also have a string resource associated with them that specifies an XPM pixmap or X11 bitmap file in exactly the same way as for the palette buttons.

```
visu*toolbar.em_copy_file.toolbarPixmap: em_copy_file.xpm
```

These pixmaps can be overridden by changing the resource or by providing a file earlier in the search path.

Makefile Features

This section describes resources that control Makefile generation. For an introduction to this feature, see Chapter 19, “Makefile Generation”, starting on page 553.

File Suffixes

Object and executable file names are derived from source file names by suffix substitution. The suffixes are specified by the following application resources:

```
visu.objectFileSuffix: .o
visu.executableFileSuffix:
visu.uidFileSuffix: .uid
```

Makefile Template

The template used for generating new Makefiles is defined in two ways: either by filename or directly in the resource file. The second way is used as a fallback if the file cannot be found.

The template file is specified by one of two resources:

```
visu.motifMakeTemplateFile: $VISUROOT/make_templates/motif[1]
visu.mmfcMakeTemplateFile: $VISUROOT/make_templates/mfc
```

There are two resources so that you can have different templates customized to pick up the appropriate class libraries. The value for the resource can contain environment variables which will be expanded by /bin/sh.

The fallback template is specified by the *makefileTemplate* resource:

```
visu.makefileTemplate:\
# System configuration\n\
# ---------------------\n\
\n\
# everything is in /usr/include or /usr/lib\n\
XINCLUDES=\n\
XLIBS=\n\
LDLIBS=\n\
.
.
```

You can edit the Makefile variables set at the beginning of the template to reflect your system configuration. For example, you can set the *XINCLUDES* variable to the path for your X include files.

Template Protocol

This section explains the symbols used in the Makefile template. In general we recommend that you not edit the default template except for the variables at the beginning. If you want to edit the actual template lines, first read Chapter 19, “Makefile Generation”, starting on page 553, and try to get the results you want by setting a variable.

Lines in the Makefile template that begin with #Sun WorkShop Visual*:* are template lines. When Sun WorkShop Visual generates or updates a Makefile, it creates instances of the appropriate template lines for each design file based on the files you have generated and converts the special symbols beginning with an *XDG_* prefix (Sun WorkShop Visual generated) to file names. *XDG_* symbols convert to a single file, or a list of files if the symbol name ends with the *_LIST* suffix.

List symbols are used singly and not mixed with ordinary symbols in lines such as the following:

```
#Sun WorkShop Visual:XD_C_PROGRAMS=XDG_C_PROGRAM_LIST
```

[1] $VISUROOT is the path to the root of the Sun WorkShop Visual installation directory.

The *XDG_C_PROGRAM_LIST* symbol translates to a list of all executables that the Makefile can build. A typical instance of this template line is:

```
#DO NOT EDIT >>>

XD_C_PROGRAMS=\
myapp1\
myapp2

#<<< DO NOT EDIT
```

Ordinary template symbols, without the *_LIST* suffix, represent single files. You can combine any number of ordinary template symbols in lines such as:

```
#Sun WorkShop Visual:XDG_C_PROGRAM: XDG_C_PROGRAM_OBJECT
$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS)

#Sun WorkShop Visual:$(CC) XDG_C_DEBUG_FLAGS $(CFLAGS) $(CPPFLAGS)
$(LDFLAGS) -o XDG_C_PROGRAM XDG_C_PROGRAM_OBJECT $(XD_C_OBJECTS)
$(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)
```

When Sun WorkShop Visual generates the Makefile, it adds a separate instance of these lines for each design file for which you have generated code with the “Main program” toggle set. Other files in the application are linked in as *XD_C_OBJECTS* and *XD_C_STUB_OBJECTS*.

```
#DO NOT EDIT >>>

myapp1: myapp1.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(CC) $(CFLAGS)
$(XDG_DEBUG_FLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp1 myapp1.o

$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)

#<<< DO NOT EDIT

#DO NOT EDIT >>>

myapp2: myapp2.o $(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(CC) $(CFLAGS)
$(XDG_DEBUG_FLAGS) $(CPPFLAGS) $(LDFLAGS) -o myapp2 myapp2.o

$(XD_C_OBJECTS) $(XD_C_STUB_OBJECTS) $(MOTIFLIBS) $(LDLIBS)

#<<< DO NOT EDIT
```

The following table shows the ordinary template symbols. You can add a *_LIST* suffix to most of these symbols to generate the corresponding list symbol. You cannot add *_LIST* to the “FLAGS” symbols. The filenames shown in parentheses are for illustration only.

TABLE 25-1 Makefile Template Symbols

Name	Use
XDG_C_PROGRAM_SOURCE	C source for main program (*main.c*)
XDG_C_PROGRAM_OBJECT	Corresponding object (*main.o*)
XDG_C_PROGRAM	Corresponding executable (*main*)

TABLE 25-1 Makefile Template Symbols *(Continued)*

Name	Use
XDG_C_SOURCE	C source (*foo.c*)
XDG_C_OBJECT	Corresponding object (*foo.o*)
XDG_C_STUB_SOURCE	C stubs (*stubs.c*)
XDG_C_STUB_OBJECT	Corresponding object (*stubs.o*)
XDG_C_EXTERN	C header (*externs.h*)
XDG_C_PIXMAP	C pixmaps (*pixmaps.h*)
XDG_CC_PROGRAM_SOURCE	C++ source for main program (*main.cpp*)
XDG_CC_PROGRAM_OBJECT	Corresponding object (*main.o*)
XDG_CC_PROGRAM	Corresponding executable (*main*)
XDG_CC_SOURCE	C++ source (*foo.cpp*)
XDG_CC_OBJECT	Corresponding object (*foo.o*)
XDG_CC_STUB_SOURCE	C++ stubs (*stubs.cpp*)
XDG_CC_STUB_OBJECT	Corresponding object (*stubs.o*)
XDG_CC_EXTERN	C++ header (*externs.h*)
XDG_CC_PIXMAP	C++ pixmaps (*pixmaps.h*)
XDG_JAVA_WIDGET_HDR	Include directory for extra Java Widgets
XDG_JAVA_WIDGET_LIB	Archive Library for extra Java Widgets
XDG_UIL_SOURCE	UIL source (*foo.uil*)
XDG_UIL_OBJECT	Corresponding compiled UIL (*foo.uid*)
XDG_C_FOR_UIL_PROGRAM_SOURCE	C for UIL source with main program (*main.c*)
XDG_C_FOR_UIL_PROGRAM_OBJECT	Corresponding object (*main.o*)
XDG_C_FOR_UIL_PROGRAM	Corresponding executable (*main*)
XDG_C_FOR_UIL_SOURCE	C for UIL source (*foo.c*)
XDG_C_FOR_UIL_OBJECT	Corresponding object (*foo.o*)
XDG_C_FOR_UIL_EXTERN	C for UIL header (*externs.h*)
XDG_UIL_PIXMAP	UIL pixmaps (*pixmaps.uil*)

TABLE 25-1 Makefile Template Symbols *(Continued)*

Name	Use
XDG_X_RESOURCE_FILE	X resource file (*foo.res*)
XDG_C_DEBUG_FLAGS	C compiler debugger flag(s)
XDG_CPP_DEBUG_FLAGS	C++ compiler debugger flag(s)

For completeness, the following table shows you the Makefile template symbols used for Smart Code generation. Note, however, that these are for Sun WorkShop Visual's private use only and are not intended to be modified.

TABLE 25-2 Private Makefile Template Symbols for Smart Code

Name	Use
XDG_CC_GROUP_OBJECT	Smart Code C++ object files
XDG_CC_GROUP_SOURCE	Smart Code C++ source files
XDG_CC_SERVER_GROUP_OBJECT	Server-side Smart Code C++ object files
XDG_CC_SERVER_GROUP_SOURCE	Server-side Smart Code C++ source files
XDG_CC_SERVER_PROGRAM	Server-side Smart Code C++ executeable name
XDG_CC_SERVER_PROGRAM_OBJECT	Server-side Smart Code C++ main object file
XDG_CC_SERVER_PROGRAM_SOURCE	Server-side Smart Code C++ main source file
XDG_C_GROUP_CFLAGS	Compiler flags required for C/C++ Smart Code
XDG_C_GROUP_LFLAGS	Linker flags required for C/C++ Smart Code
XDG_C_GROUP_OBJECT	Smart Code C object files
XDG_C_GROUP_SOURCE	Smart Code C source files
XDG_C_SERVER_GROUP_CFLAGS	Server-side Smart Code C/C++ Compiler flags
XDG_C_SERVER_GROUP_LFLAGS	Server-side Smart Code C/C++ Linker flags
XDG_C_SERVER_GROUP_OBJECT	Server-side Smart Code C object files
XDG_C_SERVER_GROUP_SOURCE	Server-side Smart Code C source files
XDG_C_SERVER_PROGRAM	Server-side Smart Code C executeable name
XDG_C_SERVER_PROGRAM_OBJECT	Server-side Smart Code C main object file
XDG_C_SERVER_PROGRAM_SOURCE	Server-side Smart Code C main source file

Dynamic Display Window

To ensure that your application looks and feels the same inside Sun WorkShop Visual and when running as an independent application, the dynamic display runs off a separate X resource database and has its own application class name - XDdynamic.

This means that the dynamic display cannot pick up Sun WorkShop Visual specific resources. What you see inside Sun WorkShop Visual is what you will see when you compile and run your code.

The dynamic display can be configured independently by creating a file called `XDdynamic` and placing both application specific and general resources in the file. For example, to produce a black and white appearance for printing add lines of the form:

```
XDdynamic*foreground: black
XDdynamic*background: white
```

For application specific resources, use the application class name as it appears in XDesigner for your product, for example:

```
XApplication*XmPushButton.background: #dededededede
```

Application specific resources of this type only load if the current application class matches the class in the file.

The XDdynamic file should be placed where Sun WorkShop Visual can pick it up through the normal X resource search mechanisms, as mentioned in “Introduction” on page 867.

XDdynamic resources affect only the dynamic display, and have no effect on any generated code.

Any Loose Bindings for the application are loaded after the XDdynamic resource file, and are similarly loaded into the separate application resource database: these take precedence over XDdynamic resources.

Specific resources applied to individual widgets through the resource panels are unaffected by all this, and will always take effect.

CHAPTER **26**

Command Summary

Introduction

This chapter is a quick reference guide to Sun WorkShop Visual commands. It includes:

- All menu commands in the main menu bar
- Some additional Sun WorkShop Visual commands
- Instructions for accessing on-line help
- A table of keyboard accelerators

This chapter is designed as a quick reference for experienced users. More detail on the commands is found in the appropriate chapters of this document and in the online help.

Some menu items execute immediately, while others display a dialog where you must enter further information before anything happens. Commands that display a dialog have three dots (...) after the command name.

Many commands have keyboard accelerators which allow you to execute them with a single keystroke. The accelerators are listed in a table at the end of this chapter.

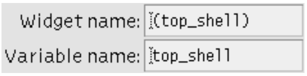
Widget Name and Variable Name

Two text boxes at the top of the main Sun WorkShop Visual screen let you specify a variable name and widget name for the currently selected widget.

The variable name is the name used to identify that widget in the generated code. Variable names must be unique. If you do not specify a variable name, Sun WorkShop Visual assigns a unique name of the form <*widget-class*><*n*>. The number *n* is not guaranteed to stay the same every time you open the design file. Therefore, you should never refer to a default name in your code.

Explicitly named widgets are global in scope. Those not explicitly named are local, unless you change this status on the Core resource panel.

The widget name can be the same as the variable name or different. Widget names do not have to be unique in a design. Groups of widgets that share a widget name also share any resource settings that are generated into the X resource file. See Chapter 7, “Generating Code”, starting on page 207, for more information.



Widget name: (top_shell)

Variable name: top_shell

FIGURE 26-1 Text Fields for Widget and Variable Name

The File Menu

Commands on Sun WorkShop Visual’s File Menu control only design files. Design files are the principal Sun WorkShop Visual files and, by convention, have the suffix *.xd*. They contain a design hierarchy (which may consist of multiple dialogs), the resource values set on the resource panels, callbacks and links.

New

Clears the construction area so that you can begin a new design. If you have not saved your work since the last change, you are asked if you want to save before the construction area is cleared.

Open...

Opens an existing design file. You are prompted for the name of the file. If you have not saved your work since the last change, you are asked if you want to save before the new file is read in to replace it.

Read...

Merges the contents of a file into your current design. Since all design files begin with a Shell widget, this adds one or more dialogs to your design.

Variable names of widgets must be unique across the entire design. When you combine two designs with “Read”, any variable names in the file that duplicate names already in your design are silently removed and replaced with local names of the default form *<widget-class><n>*.

To merge parts of dialogs from one design to another, use the “Copy to File” and “Paste from File” commands in the Edit Menu.

Save

Saves the current design using its previously specified filename. If your current design is new and has never been saved before, you are prompted for a filename as in “Save as...”.

Save as...

Saves the current design under a specified filename. The Sun WorkShop Visual file browser is displayed for you to specify a filename. Use this command when saving a new design file for the first time or if you wish to save your design in a format suitable for reading into Visaj, the visual application builder for Java. See “Moving Sun WorkShop Visual Designs to Visaj” on page 347 for more information.

Print...

Prints the current design, or selected part of it, to a printer or a file. To print to a file, click on the “File” toggle and enter the filename in the text box under “File”. To send to a printer, click on the “Command” toggle and enter the command, such as *lpr*, in the same text box, which is now labeled “Command”. Because the output is Postscript, a Postscript printer or viewer is required.

The option menus in the Print Dialog let you specify the page size, orientation, pages and scale. In the “Scale” option menu, the reduced scale option prints the diagram two-thirds of its actual size. Note that if the “Scale to fit” option is not selected, the diagram prints on as many pages as required. The “Pages” option menu lets you print all the hierarchies in your design if your design contains more than one dialog, or just the hierarchy currently displayed in the construction area.

Selecting the “Show names” toggle lets you print the variable names of the widgets. Selecting the “Print headings” toggle puts a border around the hierarchy and prints a title, which you can specify in the “Title” text field. The title can have only a single line of text.

Exit

Leaves Sun WorkShop Visual. If you have not saved your work since the last change, you are asked whether you want to save before Sun WorkShop Visual exits.

The Edit Menu

The Edit Menu has commands for editing the design hierarchy. All Edit options operate on the currently selected widget including all its children in the hierarchy.

Cut

Removes the currently selected widget from the design hierarchy and copies it into the Sun WorkShop Visual clipboard.

Copy

Copies the currently selected widget to the clipboard.

Paste

Copies the contents of the clipboard into the hierarchy as a child of the currently selected widget.

If the clipboard is empty, or if the widget in the clipboard cannot be made a child of the currently selected widget, then “Paste” is disabled.

Clear

Deletes the currently selected widget. A confirmation dialog is displayed. Cleared widgets are not put into the clipboard and therefore cannot be pasted.

Copy to File...

Copies the currently selected widget to a clipboard file. You are prompted for the file name you want to use.

Paste from File...

Copies the contents of a clipboard file into the hierarchy as a child of the currently selected widget. You are prompted for the name of the clipboard file. If the widget at the root of the clipboard file hierarchy cannot be made a child of the currently selected widget, this operation shows an “Invalid hierarchy” error message.

“Paste” always makes the pasted widget the last child of the selected widget. To reorder children of a widget, use dragging, discussed below.

Search...

Displays the search dialog. This is explained fully in the *Search section on page 2-38*.

Move by Dragging

You can move a widget to another position in the hierarchy by dragging its icon with mouse button 1. This is the equivalent of using “Cut” followed by “Paste”.

Copy by Dragging

You can copy a widget to another location in the tree by dragging its icon with mouse button 2. This is the equivalent of using “Copy” followed by “Paste”.

The View Menu

The View Menu has commands that affect the appearance of the main Sun WorkShop Visual window. Each command in this menu is a toggle and can be turned on or off. The View options only affect the appearance of the Sun WorkShop Visual screen, not your design.

Show Variable Names

Displays the name of each widget in your design underneath the widget icon in the design area. The name shown is the unique variable name of the widget.

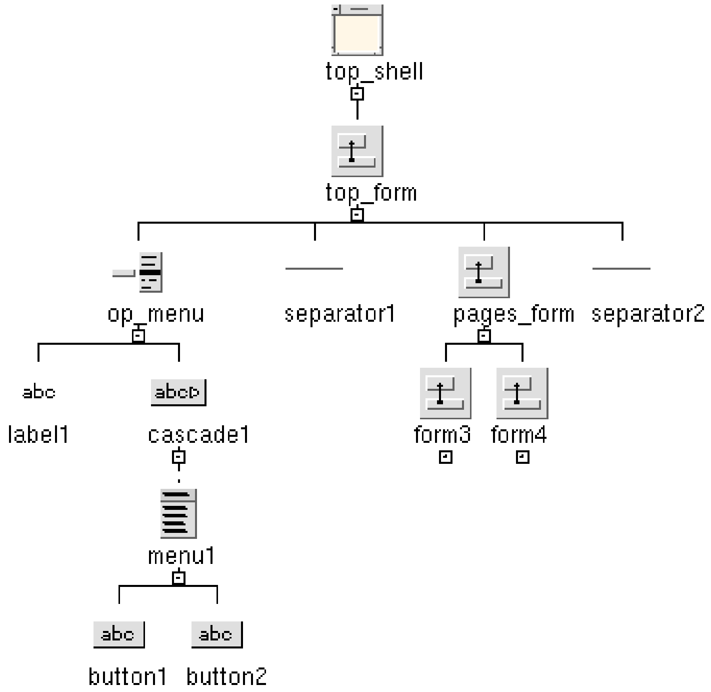


FIGURE 26-2 Show Widget Names

Show Dialog Names

Displays the widget name assigned to each Shell in your design underneath its icon in the window holding area. “Show dialog names” is particularly useful for navigating in designs that have multiple dialogs. Note that the Shell icon shrinks to allow room for the dialog name.

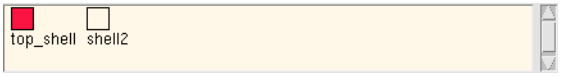


FIGURE 26-3 Show Dialog Names (Window Holding Area Shown)

Left Justify Tree

Changes the appearance of the hierarchy in the design area from a centered tree with branches spreading in both directions to a left-justified one with its branches spreading to the right.

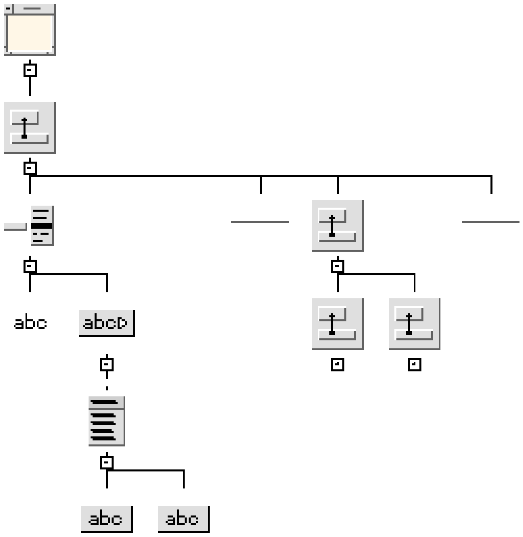


FIGURE 26-4 Left Justify Tree

Shrink Widgets

Reduces the size of widgets in the construction area. Use this option when the hierarchy you are building becomes large and you want to see the whole or a large proportion of the hierarchy. The widgets are shrunk to a uniform small square. As with the other View options, your actual design is not affected.

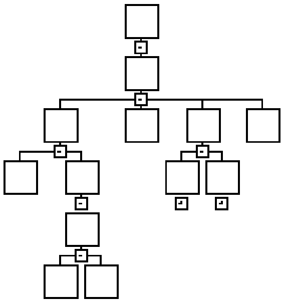


FIGURE 26-5 Shrink Widgets

Widget Annotations

Annotates widgets in the construction area according to specified criteria. You can request that the tree be annotated if the widget has:

- Callbacks
- Methods
- Links
- A pre-create prelude
- A pre-manage prelude

You can also request that the tree be annotated to indicate the results of the previous search operation. To turn on an annotation, select the appropriate toggle from the Annotations pullright menu. You can tear off the pullright menu to use as an annotation reference, as shown in Figure 26-6. To tear off the menu, click on the dashed line. Annotations are explained more fully in “Widget Annotations” on page 45.

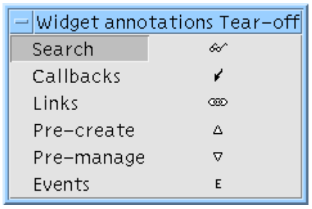


FIGURE 26-6 “Annotations” Tear-Off Menu

Structure Colors

Color-code widgets in the construction area. When this option is on, separate colors are used to indicate those widgets in the hierarchy which have been designated as functions, data structures, C++ classes, or children only (via the “Code generation” page of the Core resource panel). These designations are discussed in “Structure Colors” on page 47. This option has no effect if all your widgets are default structures.

To turn on this option, click on the “Show colors” toggle in the pullright menu. You can tear off the pullright menu to use as a color reference, as shown in Figure 26-7. To tear off the menu, click on the dashed line.

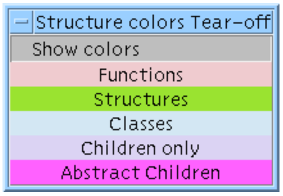


FIGURE 26-7 “Structure Colors” Tear-Off Menu

The Palette Menu

The Palette Menu contains options which change the appearance of the Widget Palette.

Layout

Specifies whether the Palette is to be displayed with icons, labels, or both. Click on the appropriate toggle button in the pullright menu.

Separate Palette

Displays the Widget Palette in a separate sub-dialog. Sun WorkShop Visual Window will then have more room for displaying the Design Area. You can select from and add to the separated Widget Palette in the normal manner.

Show Palette

Re-displays and raises the separated Widget Palette to the top of the window stack. This is a useful option when you have closed down the separated Widget Palette, or have lost it underneath all your separate application dialogs. This option is disabled if a separate palette has not been chosen.

Define

Designates the currently selected widget as a widget definition. This option is a short-hand method of creating widget definitions. The currently selected widget (if specified as a class) will form a new definition using some default configuration data deduced by context.

Edit Definitions...

Displays the Edit Definitions panel which allows you to turn portions of your widget hierarchy into reusable objects selectable from the Widget Palette.

The Widget Menu

The Widget Menu has commands that apply to individual widgets. All these commands apply to the currently selected widget in the hierarchy.

Resources...

Displays the resource panel for the currently selected widget. You can also display the resource panel for most classes of widgets by clicking twice on the widget's icon in the design hierarchy.

Resource panels are discussed in detail in *Chapter 3, "Resources", starting on page 53*. Some individual resources are also discussed on a per-widget basis in Chapter 27, "Widget Reference", starting on page 743.

Core Resources...

Displays the Core resource panel, where you can set resources inherited from the Core, Primitive and Manager superclasses. Core resources include foreground and background colors and whether or not a widget is sensitive to events.

The “Code Generation” page of the Core resource panel lets you specify individual widgets as local, global, or static, regardless of whether they are explicitly named. If you are using C++, this page also lets you specify public, private, or protected status for the selected widget. It also allows you to change the class access for Java code generation.

Loose Bindings...

Displays the Loose Bindings Dialog. This allows you to control the way in which resources are generated in the resource file so that widgets can share resources. See “Loose Bindings” on page 86 for more details.

Layout...

Displays the Layout Editor. This option is available for the four classes of layout widget - the Form, BulletinBoard, RowColumn and DrawingArea. For more information, see Chapter 4, “The Layout Editor”, starting on page 97.

Constraints...

Displays the constraints panel for any widget that is a child of a constraint widget - a PanedWindow or Form. In the case of the Form, this panel should usually be used only to view the constraint resources rather than to reset them, because Form attachments can be set more reliably in the Layout Editor. The Constraints panel is discussed in *Chapter 3, “Resources”, starting on page 53*.

Callbacks...

This displays the Callbacks Dialog, as shown in Figure 26-8. You can set callbacks for the selected widget here. This dialog is described in detail in “Designating a Callback” on page 173.

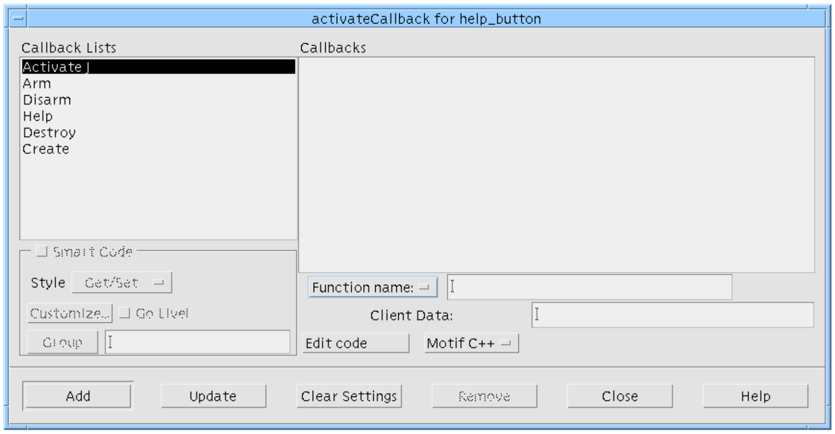


FIGURE 26-8 Callbacks Dialog

Event Handlers...

Displays the Event Handlers Dialog which allows you to add low level input handling for a widget, bypassing the widget’s translation tables. Event handlers are generated by Sun WorkShop Visual into the code file as part of the general widget configuration and a stub is generated into the stubs file. See “Event Handlers” on page 203 for more information.

Translations...

Lets you specify translations for the currently selected widget. A translation is a key sequence that is mapped to a specified action on the widget. Translations can be one of two kinds, “Override” or “Augment”. “Override” translations override any translations already set on the given key sequence. “Augment” translations are added to the list of translations already set.

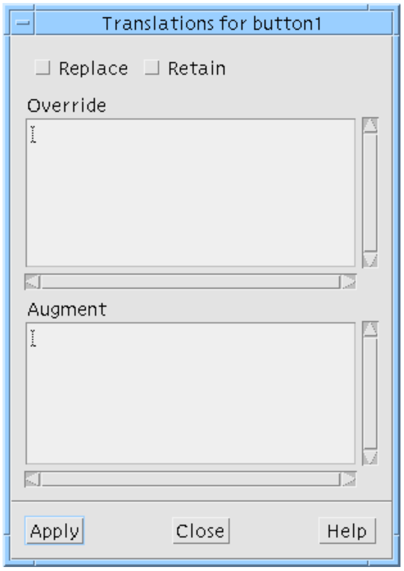


FIGURE 26-9 Translations Dialog

If you use the “Replace” toggle on this dialog, the translations you type into the upper box replace any translations already set on this widget. When “Replace” is set, you cannot use the “Augment” text box.

For more information, see “Translations and Actions” on page 190.

Add to Groups...

Displays the Group Editor where you can add the selected widget to an existing Group. Groups are the nuts and bolts of Smart Code which, in turn, provides the foundations for creating applications capable of connecting to the World Wide Web.

The Group Editor also allows you to edit the Groups. Groups, along with the Group Editor, are explained in Chapter 15, “Groups”, starting on page 477. “Get/Set Tutorial” on page 493 contains a short tutorial to help you get the most from Groups and basic Smart Code.

Add to a New Group...

Creates a new Group adding the selected widget as its first member. This option displays the Group Editor, allowing you to edit any existing Groups, including the one you have just added, as described in the above section.

Code Preludes...

Allows you to add pieces of code to the code generated for the currently selected widget. The most commonly used types are pre-create and pre-manage preludes. Pre-create preludes appear just before the widget is created. A typical use of pre-creation preludes is to set widget resources that can only be set at widget creation time.

Pre-manage preludes appear just before the widget's callbacks are added. They are typically used for setting up client data for the callbacks.

When you select "Code preludes" you are given the choice of editing "in place" (in the generated code) or typing code into a dialog and allowing Sun WorkShop Visual to add it to the generated code. If you choose to "Edit in place", the generated code is opened at the appropriate line for the type of prelude selected. See "Code Preludes" on page 242 for more details on prelude editing. See "Setting up Callback and Prelude Editing" on page 701 for details on using the edit mechanism.

Code preludes for private, protected and public methods apply to C++ class widgets. For more information, see "Adding Class Members" on page 291.

Method Declarations

Allows you to add, remove and edit method declarations in a widget's C++ class.

Reset

Destroys the currently selected widget and all its children and recreates them. This is useful after you set certain resources on the resource panels or in the Layout Editor. Sometimes creating a widget and then changing a resource value gives a different result from creating the widget initially with the changed value. Therefore, if your dynamic display does not reflect changed resource settings the way you expected, resetting widgets can often solve the problem.

Edit Links...

Allows you to you set up or remove links from widgets. Links are pre-defined Activate callbacks and can be set only on button-type widgets. Links can show, hide, manage, unmanage, enable or disable any widget. You can set multiple links on one button. For more information see "Links" on page 183.

Fold / Unfold

Hides / displays the children of a selected widget. Use this toggle to save space when your design hierarchy becomes too large to fit in the construction area. When the selected widget is folded, its children are hidden, though they are not removed from the design. When a widget is folded, the fold icon changes to a plus sign (+). When it is unfolded, the icon is a minus sign (-). You can also click over this icon to fold or unfold the hierarchy.

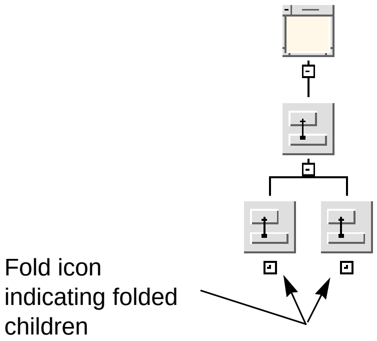


FIGURE 26-10 Folded Widgets

Definition

Marks the currently selected hierarchy of widgets as a reusable object definition. The definition can be inserted into the widget palette and then selected just like any other palette widget.

The Module Menu

The commands on this menu refer to operations on the whole design rather than individual widgets.

Loose Bindings...

Displays the Loose Bindings dialog, allowing you to set loose resource bindings for the whole design. See “Loose Bindings” on page 86 for a description of the Loose Bindings dialog.

Module Preludes...

Displays a dialog box that lets you enter lines of code to be entered at or near the beginning of the generated code file.

The Module Heading is inserted at the beginning of the primary code module and at the beginning of the stubs file, if generated. The Module Heading is typically used for SCCS ids, versions and other identifying information.

The Module Prelude is inserted after the generated Sun WorkShop Visual *#include* directives, if any, and is typically used for *#include* or *#define* directives or *extern* declarations required by your code preludes.

The Resource Prelude is inserted after the Sun WorkShop Visual generated comment in the X resource file. It can be used to set global application resources or to *#include* another resource file.

Help Defaults

Displays a dialog which allows you to specify defaults for the help system.

The Default Path field denotes the path used for help document names. It is also used in Sun WorkShop Visual as a fall-back location for generated code.

The Default Document field is used if a marker for help is set but no document is specified for the widget or any of the widget ancestors.

The Path Resource and Path environment variable fields allow you to provide a dynamic help document context: code will be generated so that any values specified will be used to override any setting of the Default path: the runtime environment variable takes precedence over the static resource setting.

The Default Translation field will cause a translation to be added to every widget which has a help marker; the associated Action for this translation will always be Help. For example, a translation ‘Ctrl<key>A’ will mean that unless specified otherwise Control A will cause the Help message window to be displayed for a widget which has a help marker.

The ‘Always own window’ toggle, when set, causes code to be generated so that each access to the Help System will result in a new Help Document window.

See Chapter 21, “Hypertext Help”, starting on page 597 for more information on adding help to your design.

Work Procedures...

Displays a dialog allowing you to add the names of functions which you wish to be called when the X toolkit has no other events to process. X work procedures are, therefore, a convenient means of setting up a background batch process. Stubs are generated for these procedures in exactly the same was as for widget callbacks. “Xt Work Procedures” on page 200 provides a more thorough description of this process.

Input Procedures...

Displays a dialog for adding the names of functions you wish to be called when a specified file or pipe is ready for reading or writing. This allows you an alternative source for events. More about these is given in “Input Procedures” on page 201.

Language Procedure...

Displays a dialog where you can specify the name of a language procedure which is called when your application starts up. This gives you a place to provide additional code for setting up the locale. See “Language Procedures” on page 202 for more information on this topic.

Timeout Procedures...

Displays a dialog where you can add the name of a procedure to be called after a specified amount of time has elapsed. The stubs for these are generated into the main code file. See “Timeout Procedures” on page 201 for more information.

Action Procedures...

Allows you to specify the action associated with a particular translation. See “Translations and Actions” on page 190 for information on setting up a translation for a widget and “Additional Actions” on page 197 for information on adding an action procedure.

Groups...

Displays the Group Editor where you may edit the Groups in your design. Groups are the nuts and bolts of Smart Code which, in turn, provides the foundations for creating applications capable of connecting to the World Wide Web. Groups, along with the Group Editor, are explained in Chapter 15, “Groups”, starting on page 477. “Get / Set Tutorial” on page 493 contains a short tutorial to help you get to grips with Groups and basic Smart Code.

Java Compliant

When this toggle button is selected, Sun WorkShop Visual checks over your design to see whether the design can be generated as Java code. If the compliance check fails, a dialog appears informing you of the failures. Some of these can be fixed automatically by Sun WorkShop Visual, if you wish. This dialog is described in “Java Compliance Failure Dialog” on page 316.

Microsoft Windows compliant

This toggle (only present when Sun WorkShop Visual is in Microsoft Windows mode) is used to indicate that the design is Microsoft Windows compliant, i.e. it is possible to generate MFC code for it. The toggle is set if the design is compliant. If the design becomes non-compliant, the toggle is unset and the Microsoft Windows Compliance Failure dialog is displayed (see “Compliance Failure” on page 369).

Application Class...

The MFC and Motif XP flavors use an instance of the CWinApp class to represent the application. By popping up the Application class dialog you can change the base class name, the class name and the instantiate as name for this instance. This item is only present when Sun WorkShop Visual is in Microsoft Windows mode.

The Generate Menu

When your design is complete, the Generate Menu can be used to generate code for it in three languages: C, C++, or UIL. The three menu items relating to these languages are pullright menus from which you can select the file you wish to generate. From this menu you can also generate an X resource file containing resource values explicitly set in your design, a Microsoft Windows resource file (if you are running in Microsoft Windows mode) and a Makefile. You can also select a general “Generate” option to display the Generate dialog and then specify all the files to be generated. The procedure for generating code, linking and running is discussed in Chapter 7, “Generating Code”, starting on page 207.

If you select a particular file from the menu (C, C Stubs, C++ Stubs etc.), the generate dialog will only appear if you have not generated a file of that type before - otherwise the file will be generated silently.

C

Displays the C tear-off or pullright menu.

C++

Displays the C++ tear-off or pullright menu.

UIL

Displays the UIL tear-off or pullright menu.

X Resource File...

Generates an X resource file from your design. By convention, generated X resource files have the suffix *.res*. They must be copied to the location expected by X in order to take effect.

You must use the same application class name for the X resource file that you used when you generated the primary module, or X will not be able to associate your resources with the generated application.

Microsoft Windows Resources...

Generate a Microsoft Windows resource file and the associated bitmap (*.bmp*) and icon (*.ico*) files. The Microsoft Windows resource file typically has the extension *.rc*. This option is only present when Sun WorkShop Visual is in Microsoft Windows mode.

Makefile...

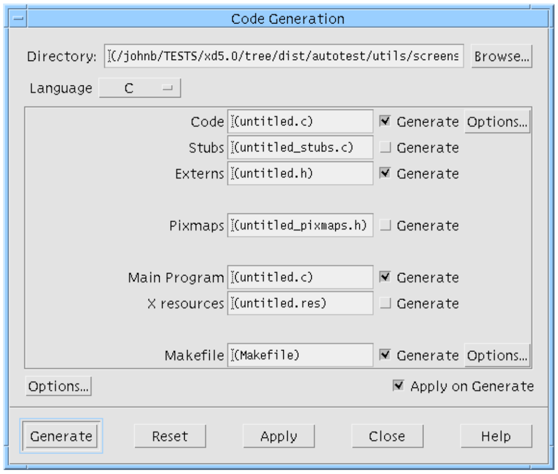
Generates a Makefile to build your application. If you have specified filenames in the Generate dialog, these are used. Otherwise the default names are used.

Java

Generates Java code for your design. The first time you select this option, the Generate dialog is displayed with the language set to “Java”, thereby allowing you to set any options you may require. Once you have generated Java code, selecting this option will generate the previously selected files without displaying the Generate dialog. Instead, a message appears telling you which files have been generated. For more information on generating Java code, see “Generate Dialog” on page 336.

Generate...

Displays the Generate dialog, as shown in Figure 26-11. This dialog can be used to generate files, to save generate option settings and to enable you to see “at-a-glance” the files that you are generating.



Code Generation

Directory: (/johnb/TESTS/xd5.0/tree/dist/autotest/utils/screens

Language: C

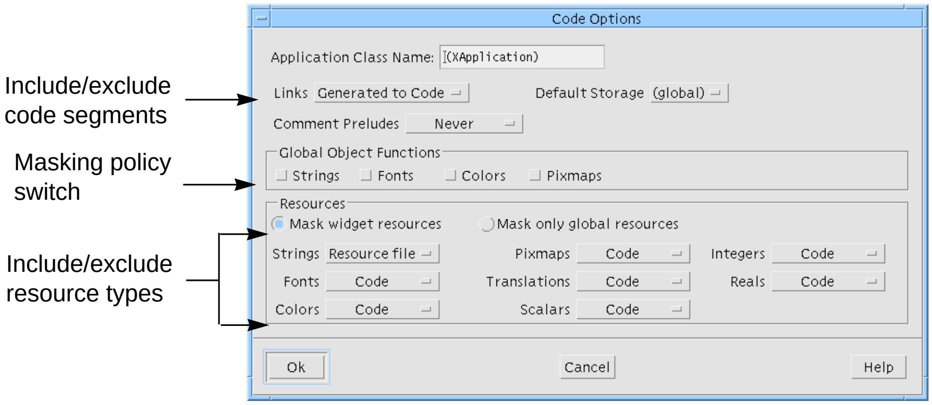
Code (untitled.c) ☑ Generate
Stubs (untitled_stubs.c) □ Generate
Externs (untitled.h) ☑ Generate
Pixmaps (untitled_pixmaps.h) □ Generate
Main Program (untitled.c) ☑ Generate
X resources (untitled.res) □ Generate
Makefile (Makefile) ☑ Generate

☑ Apply on Generate

FIGURE 26-11 Generate Dialog

The Generate dialog allows you to type in the names of the files to be generated using either the full pathname or a name relative to a base directory which you can specify in this dialog.

There are a number of sub-dialogs in the Generate dialog which relate to individual files. There is also a sub-dialog which allows you to set code options which affect code generation generally. The Code Options dialog contains sets of option menus which control generation of code segments (in code files only), types of resources (in code and X resource files) and a masking policy switch that works in conjunction with the resource type toggles. The Code Options dialog, shown in Figure 26-12, is explained in Chapter 7, “Generating Code”, starting on page 207.



Code Options

Application Class Name: (XApplication)

Links: Generated to Code
Default Storage: (global)
Comment Preludes: Never

Global Object Functions
□ Strings □ Fonts □ Colors □ Pixmaps

Resources
● Mask widget resources ○ Mask only global resources

Strings: Resource file
Fonts: Code
Colors: Code
Pixmaps: Code
Translations: Code
Scalars: Code
Integers: Code
Reals: Code

Include/exclude code segments

Masking policy switch

Include/exclude resource types

FIGURE 26-12 Code Generation Options Dialog

Tear-Off Menus

The procedure for generating code is basically the same regardless of your choice of language. The pullright menus for C, C++ and UIL can be torn off into a separate window by clicking on the dashed line at the top of the menu. You can also invoke commands from the pullright menu in the usual way without tearing off the menu. After you have displayed or torn off one of these menus, you can select one of the following commands.

C, C++, or UIL...

Generates the primary code module in the language of your choice.

Stubs or C++ Stubs...

Generates a stubs file, containing function declarations and empty braces for any callback functions you have designated in your design. Use of a stubs file ensures that your callbacks are declared with the proper syntax. You can then write code between the braces to add functionality.

Externs, C++ Externs, or Externs for UIL...

Generates a header file that declares all global objects and functions in your design. For convenient access to your global widgets and defined objects, *#include* this file in your primary code module or stubs file.

C, C++, or UIL Pixmaps...

Generates static declarations of all pixmaps in your design into a separate file. This file is meant to be included as a header file and by convention has the suffix *.h*.

C, C++, or UIL Main Module...

Generates the primary code module in the language of your choice.

C for UIL...

Generates a C file that performs those functions in your application not covered by the UIL file. This command applies only to UIL applications. It exists because UIL does not have all the capabilities offered in Sun WorkShop Visual. You must first generate a UIL file and specify the name of that file in the “Uid File” text field.

The Tools Menu

This menu contains a list of available tools. The tools fall into two categories: those that are independent but work in conjunction with Sun WorkShop Visual and those that are an integral part of Sun WorkShop Visual.

AppGuru Designer...

Displays the AppGuru Design dialog where you can choose an existing AppGuru template, edit the templates and create new ones. Selecting a template from this dialog adds it to your design, thus giving you reusable, standard interfaces. See “AppGuru” on page 418 for details.

Pixmap Editor

This is an internal editor for creating pixmaps. See “Editing Pixmaps” on page 148 for more details.

Font Selector

This is an internal tool for selecting fonts and creating font objects. See “Setting Fonts” on page 139 for more details.

Color Selector

This is an internal tool for selecting colors and creating color objects. See “Setting Colors” on page 135 for details.

Sun WorkShop Visual Capture

This is a separate tool which captures the design of a Motif/Xt application of your choice. See “Sun WorkShop Visual Capture” on page 425 for more details on capturing designs.

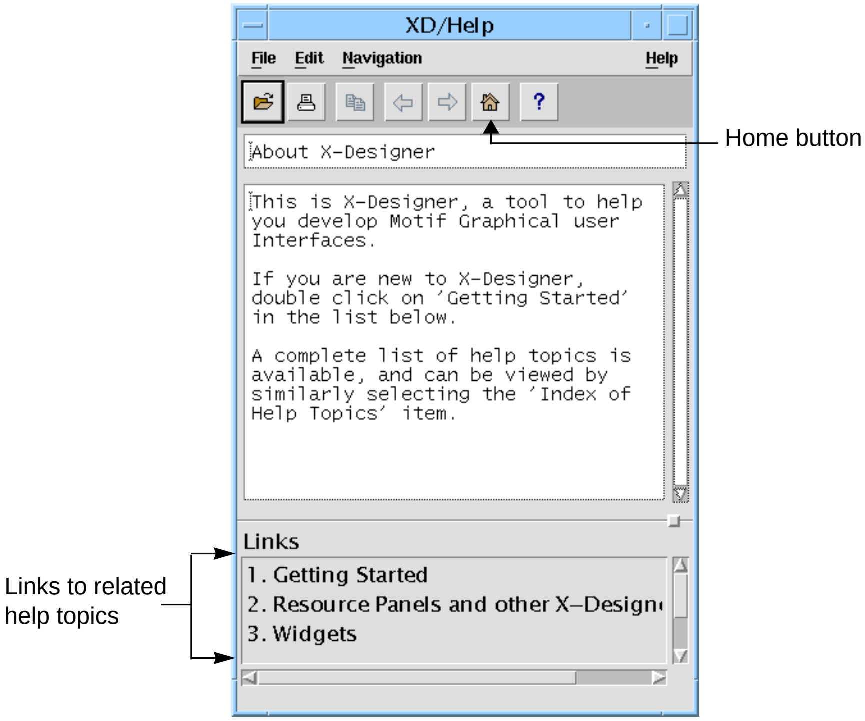
Sun WorkShop Visual Replay

This is a separate tool which allows you to record your use of a Motif/Xt application of your choice and to replay the recorded script. See Chapter 14, “Sun WorkShop Visual Replay” for more details on recording and replaying.

The Help Menu

The Help Menu provides ways of getting information about Sun WorkShop Visual.

To get specific help, click on the “Help” button on any dialog box. Sun WorkShop Visual’s help is organized as a hypertext network. Each help screen displays a list of related topics, as shown in Figure 26-13. To display help for one of the related topics, double click on the topic.



About X-Designer

This is X-Designer, a tool to help you develop Motif Graphical user Interfaces.

If you are new to X-Designer, double click on 'Getting Started' in the list below.

A complete list of help topics is available, and can be viewed by similarly selecting the 'Index of Help Topics' item.

1. Getting Started
2. Resource Panels and other X–Design
3. Widgets

FIGURE 26-13 Help Viewer

Click on the “Home” button to go to the top help screen. Double click on the “Index of Help Topics” link for a complete list of help topics.

About Sun WorkShop Visual

Displays the Sun WorkShop Visual copyright screen and the version of the software.

Palette Icons...

Displays a window with pictures of all the Motif widget icons. You can click on any of these icons to get information about that widget class. The Palette Icons window can be iconified independently of the main Sun WorkShop Visual window.

Help...

Displays the top-level help screen in the hypertext index. This screen offers a very general help message and the opportunity to follow links into more specific subjects.

Documentation...

Displays the User's Guide in Netscape. If you have selected Sun WorkShop Visual Help as the viewer, this item is unavailable.

Viewer

This is a pullright menu which allows you to choose which application to use to display Sun WorkShop Visual help. There are two items in the menu:

- Use Netscape - the popular HTML browser
- Use XD/Help - Sun WorkShop Visual's own HTML viewer

Keyboard Shortcuts

The following table lists the keyboard accelerators for the Sun WorkShop Visual commands.

TABLE 26-1 Keyboard Accelerators for Sun WorkShop Visual Commands

Menu	Command	Accelerator
File	New	Control+N
	Open...	Control+O
	Read...	Control+R
	Save	Control+A
	Save as....	Control+V
	Print...	Control+P
	Exit	Control+E
Module	Module Prelude...	Control+F9

TABLE 26-1 Keyboard Accelerators for Sun WorkShop Visual Commands *(Continued)*

Menu	Command	Accelerator
Edit	Cut	<keypad>Cut
	Copy	<keypad>Copy
	Paste	<keypad>Paste
	Clear	Del
View	Show widget names	Control+W
	Show dialog names	Control+D
	Left justify tree	Control+L
Widget	Core resources...	Control+C
	Event Handlers...	Control+H
	Reset	Control+T
	Fold / unfold	Control+F
Generate	C...	Alt+C
	UIL...	Alt+U
	C for UIL...	Alt+L
	X Resources...	Alt+X
	Makefile...	Alt+K
	Java	Alt+J
Pixmap Editor	New	Control+N
	Open	Control+O
	Save	Control+S
	Close	Control+Y
	Undo	<keypad>Paste
	Cut	<keypad>Cut
	Copy	<keypad>Copy
	Paste	<keypad>Paste
	Clear	Del
	Select All	Control+A
	Resize	Control+R
	Edit Palette	Control+P

TABLE 26-1 Keyboard Accelerators for Sun WorkShop Visual Commands *(Continued)*

Menu	Command	Accelerator
Layout Editor	Close	Control+Y
	Undo	Control+Z
	Reset	Control+T
	Widget Names	Control+W
	Class Names	Control+N
	Edge Highlights	Control+E
Help	About Sun WorkShop Visual...	Control+F10
	Help...	F1 or HELP

CHAPTER **27**

Widget Reference

Introduction

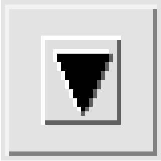
This chapter provides a brief description of each of the Motif widgets in the widget palette and gives hints for their effective use. It also describes some of the quirks of each widget. Only basic information is given here. For a full description of each widget, including all its resources, see the *Motif Programmer's Reference Manual*.

Each widget description starts with a list of resources grouped by page in the resource panel. Also listed are the callbacks applicable to the widget. Although, strictly speaking, the callbacks for a widget *are* resources, they are added, edited and viewed in a separate callbacks dialog.

Core resources are not listed to avoid repetition. Resources in **`bold`** typeface are frequently set and so are of interest to users regardless of their level of expertise. Resources in normal typeface are less commonly used and you may require more knowledge to use them effectively. Resources that are in *italics* are not applicable to that widget and are insensitive on the resource panel.

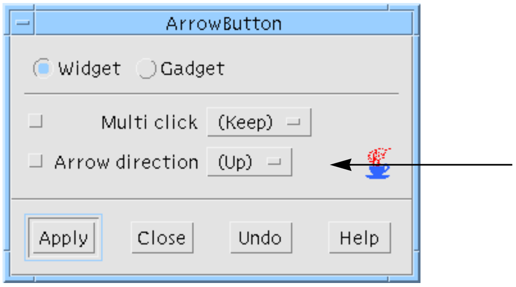
Note – If you invoke Sun WorkShop Visual using the command `small_visu`, the widget icons are smaller and slightly different from those shown here.

ArrowButton



Settings	Callbacks	Toggles
Arrow direction	**Activate**	Widget
	Arm	Gadget
	Disarm	

The ArrowButton widget provides a button with an arrow on it instead of a text label. The arrow can point up, down, left, or right. Choose one of the four directions by selecting the appropriate direction from the option menu on the ArrowButton resource panel which is shown in Figure 27-1.



ArrowButton

● Widget ○ Gadget

□ Multi click (Keep)

□ Arrow direction (Up)

Select direction

FIGURE 27-1 ArrowButton Resource Panel

Unlike other button widgets, ArrowButtons are not derived from the Label and cannot display text or pixmap labels.

BulletinBoard

Display

Title
Margin height
Margin width
Horizontal spacing
Vertical spacing
Fraction base
Cancel button
Default button

Settings

Dialog style
Resize policy
Shadow
Allow overlap
Auto unmanage
Default position
No resize
Rubber positioning

Fonts

Text font
Button font
Label font

Callbacks

Focus
Map
Unmap

The BulletinBoard widget is the most basic container widget. It is most commonly used internally by Motif to implement other container or composite widgets such as the Form, SelectionBox and MessageBox. These derived widgets are often more useful than the BulletinBoard itself.

As a container widget, the BulletinBoard does not impose any particular layout on its children. It provides absolute positioning, margin constraints and lets you specify whether the widgets inside are allowed to overlap or not. Resizing the BulletinBoard does not move or resize the widgets in it.

The BulletinBoard is most useful for transient dialogs that are not meant to be resized. For resizable dialogs, use a Form or DialogTemplate. You can also use a BulletinBoard for cases where complicated positioning is required and code is to be written for this purpose.

Only the Move and Resize options of the Layout Editor can be used with a BulletinBoard. Attachments between widgets and position attachments are not available. For more flexible layout options, use a Form widget.

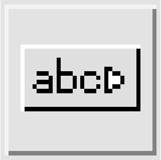
If a BulletinBoard is the child of a Shell, the BulletinBoard’s “Title” resource is used as the title of the Shell’s window.

Note that the “Title”, “Dialog style”, “Default position” and “No resize” resources are disabled if the BulletinBoard is a child of the Form.

The “Auto unmanage” resource, when set to “Yes”, makes the dialog disappear whenever you click on a button child of a BulletinBoard. This behavior, which is the default behavior in Motif, is useful for transient dialogs but can be confusing in main windows. Sun WorkShop Visual explicitly sets this resource to “No” for the BulletinBoard and two of its derivatives, the DialogTemplate and the Form. With other BulletinBoard derivatives, Sun WorkShop Visual does not override the default "Yes" setting and so the dialog does disappear if you click any button. To restore your dynamic display, reset the Shell or select the Shell icon in the window holding area.

The "Default position" resource controls how the position of the window on the screen is determined. If you set this resource to "No" on a BulletinBoard (or derivative) that is a child of a Shell, the window is displayed in the position determined by the *x* and *y* resources of the BulletinBoard, not those of the Shell. As this behavior is dependent on the window manager, it may not be consistent.

CascadeButton



Display

Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select pixmap
Select insensitive pixmap

Margins

Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Toggles

Widget
Gadget

Keyboard

Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Callbacks

Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Settings

Alignment
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

The CascadeButton widget is used to display a menu. A CascadeButton can only be used as the child of a MenuBar or Menu. When it is the child of a MenuBar, a pulldown menu is displayed and when it is the child of a Menu, a pullright menu is displayed. Sample hierarchies showing these specific uses of the CascadeButton are located in the Menu widget description.

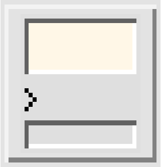
The only permissible child of a CascadeButton is a Menu. When a user clicks on a CascadeButton, a menu is displayed.

In Motif, a Menu is not technically a child of a CascadeButton but of the button's parent. To indicate this, the connection between a CascadeButton and its menu is drawn with a dotted line instead of the normal solid line.

You can set keyboard mnemonics for CascadeButtons to let the user navigate through the menus without using the mouse.

Note that the Mapping delay resource can be used only when the Button is used to instigate a pullright menu.

Command



Display

No match string
Pattern
Max history items
Command
History item count
Text columns
Directory mask
Directory

Labels

Apply label
OK label
Cancel label
Help label
List label
Prompt string
Prompt string
Directory label

Settings

Dialog type
Minimize buttons
Must match
File type
Work area placement

Callbacks

Apply
Cancel
OK
No match
Command changed
Command entered

The Command widget is a composite widget used to select a command from a scrollable history list of commands. Commands can be typed into a text area at the bottom of the widget. When a command is entered, it is added to the end of the history list. A Command inherits some BulletinBoard resources. To display the BulletinBoard resource panel, click on “Bulletin Board Resources” in the resource panel.

A Command contains a ScrolledList widget for the command history region, a Label widget for the command line prompt and a Text widget for the command entry region. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. You can change the default resource settings for the component widgets but you cannot delete them. To change the prompt, change the “Prompt string” resource in the resource panel of the Command, not in the resource panel of its Label child.

A Command is usually used in a Shell or MainWindow.

You can add multiple children to a Command. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the Command widget’s hierarchy as shown in Figure 27-2. The additional children can include a MenuBar and any number of PushButton widgets.

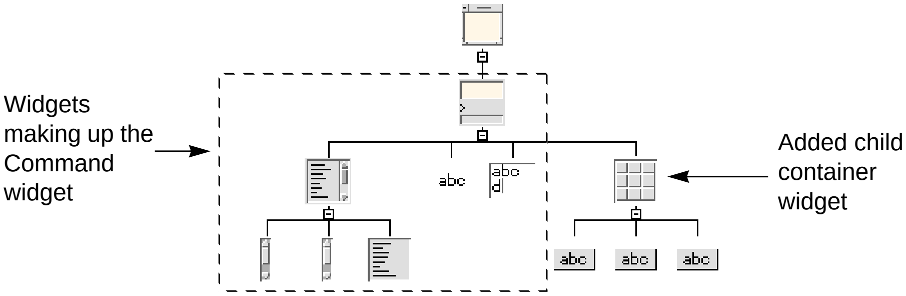

Widgets making up the Command widget
abc
abc d
Added child container widget
abc
abc
abc

FIGURE 27-2 Command Widget Hierarchy

DialogTemplate

Display

Message text
OK label
Cancel label
Help label
Symbol pixmap

Callbacks

Cancel
Ok

Settings

Default button
Dialog type
Alignment
Minimize buttons

The DialogTemplate widget is usually used as the child of a Shell for a broad range of dialogs. It provides a standard layout that includes, from top to bottom, a menu bar, a work area, a Separator, and a button box.

The DialogTemplate is a specially configured MessageBox and shares its resource panel. It also inherits some BulletinBoard resources. To display the BulletinBoard resource panel, click on “Bulletin Board Resources” in the resource panel.

The Separator is a component part of the DialogTemplate. You must add the other elements of the standard layout if you want them: for example, a MenuBar, any type of widget for the work area and buttons of any type for the button box, as shown in

Figure 27-3. The work area can be a container widget, such as a Form, with children. The DialogTemplate always arranges its children in the standard order from bottom to top, regardless of the order in which you add them to the hierarchy.

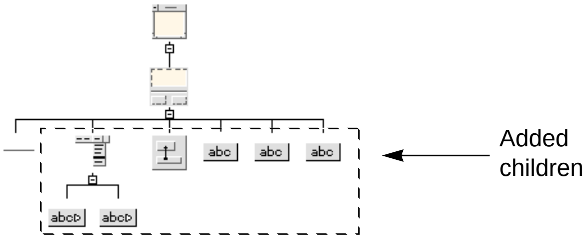


FIGURE 27-3 Standard Hierarchy Using the DialogTemplate

The areas of the standard layout are constrained to be the same width, with the buttons in the button box evenly spaced in one or more rows. The buttons are automatically rearranged as needed when the window resizes.

DrawingArea

Margins
Width
Height

Resize policy

Callbacks
Expose
Input
Resize

The DrawingArea widget provides an area in which an application can display output graphics. For example, the design hierarchy in the main Sun WorkShop Visual window is drawn in a DrawingArea contained in a ScrolledWindow.

To display the Layout Editor, select “Layout...” from the Widget menu or press the Layout button on the toolbar. Only the Move and Resize options of the Layout Editor can be used with a DrawingArea. Attachments between widgets and position attachments are not available.

To display the resource panel, select “Resources...” from the Widget pulldown menu or double-click over the widget.

Although a DrawingArea can have any number and types of children, it is not very useful for managing the geometry of other widgets. Other container widgets such as the Form should be used for this purpose instead.

Sun WorkShop Visual cannot help you with drawing in the drawing area. To do this, you must write code containing X Graphics calls. This code is normally put in the "Expose" callback.

DrawnButton

Display

Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Settings

Alignment
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

Margins

Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Callbacks

Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Keyboard

Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Toggles

Widget
Gadget

The DrawnButton widget is similar to a PushButton except that its face must be drawn by the application instead of being drawn automatically. It can be used to provide a button that has a context-sensitive appearance.

To display a picture on a button, it is usually easier to use a PushButton with a pixmap for the image. Drawing the picture on a DrawnButton requires writing code containing X graphics calls, which is normally put in the "Expose" callback.

FileSelectionBox

Display

No match string
Pattern
Max history items
Directory spec
Visible item count
Text columns
Directory mask
Directory

Settings

Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels

Apply label
OK label
Cancel label
Help label
File list label
Selection label
Filter label
Directory label

Callbacks

Apply
Cancel
OK
No match
Command changed
Command entered

The FileSelectionBox widget is a composite widget that lets users browse through the file system and select a file. The file browser in Sun WorkShop Visual is an example of a FileSelectionBox. The Generate Dialog is a FileSelectionBox with a work area child. The FileSelectionBox is derived from the SelectionBox and shares its resource panel.

The FileSelectionBox combination includes two ScrolledLists, two TextFields, four Labels, a Separator and four PushButtons, which are gadgets. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. To display the resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

While a FileSelectionBox can be used anywhere that a BulletinBoard can, it is usually placed in a Dialog Shell that is popped up for file selection.

To change the labels of button or label widgets from the defaults, change the resources in the resource panel of the FileSelectionBox, not in the resource panels of the individual widgets.

You can add multiple children to a FileSelectionBox. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the FileSelectionBox widget’s hierarchy. The additional children can include a MenuBar and any number of PushButton widgets.

Form

Display

Title
Margin height
Margin width
Horizontal spacing
Vertical spacing
Fraction base
Cancel button
Default button

Settings

Dialog style
Resize policy
Shadow
Allow overlap
Auto unmanage
Default position
No resize
Rubber positioning

Fonts

Text font
Button font
Label font

Callbacks

Focus
Map
Unmap

The Form widget is a container widget that provides both absolute and relative positioning of its children widgets. It is commonly used to lay out widgets in a dialog, either as a child of a Shell or as the work area in a DialogTemplate or similar widget.

The layout of widgets in a Form is specified by using attachments on children of the Form. Different types of attachments let you specify different types of spatial relationships such as a fixed location within the Form, a relative location within the Form, or a fixed distance between widgets. These capabilities allow considerable flexibility and reliable behavior when widgets or windows are resized. Sun WorkShop Visual lets you specify these attachments interactively in the Layout Editor. For more information, see the *Layout Editor* chapter.

You can view the attachments set on any child of a Form by using the Constraints panel. Select any child of the Form, pull down the Widget Menu and select “Constraints...”. Use of this panel is described in the *Using the Resource Panels* chapter. It is particular useful if you have to superimpose one widget over another.

The “Auto unmanage” resource, if set to “Yes”, makes the dialog erase whenever you click on a button child of a Form. This behavior, the default behavior in Motif, is useful for transient dialogs. However, because a Form is often used for main windows, Sun WorkShop Visual explicitly sets this resource to “No” in the case of the Form. Set it to “Yes” if you want a Form to auto unmanage.

For more details, see BulletinBoard.

Frame

Display

Margin width
Margin height
Title widget
Title spacing
Shadow type
Title alignment (horizontal)
Title alignment (vertical)

The Frame widget is used to provide a border, possibly with a title, around a widget that otherwise has none, to enhance the border of a widget that already has one, or to create a border around a group of widgets. A Frame can be used to provide three-dimensional effects, like indenting a DrawingArea.

A Frame can have two children. The first is placed inside the Frame and the second (which is optional) is used as a title. The second child is usually a Label.

To create a border around a group of widgets, they must be placed in a container widget such as a RowColumn or a Form, that is the child of a Frame, as shown in Figure 27-4.

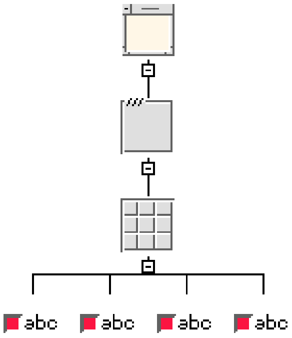
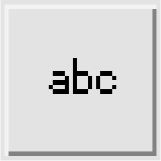


FIGURE 27-4 Hierarchy Showing a Frame Widget as a Border

The Frame sizes itself to match the size of its children.

Label



Display

Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Settings

Alignment
Type
Resize
Push button
Shadow in
Fill on arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

Margins

Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Callbacks

Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Keyboard

Accelerator
Accelerator text
Mnemonic
Mnemonic charset
Mapping delay

Toggles

Widget
Gadget

The Label widget provides a static display area for text or pixmap images. Labels are commonly used to display descriptive text strings or icons or logos. Labels can be placed in menus to provide unselectable titles for groups of menu items.

A string in a Label can extend over multiple lines and have multiple fonts. Multiple fonts are supported using the Compound String Editor, which is discussed in “Compound Strings” on page 163.

If you set a pixmap for a Label, the Label does not display it until you also change its “Type” setting to “Pixmap”.

List

Display

Margin width
Margin height
Spacing
Visible items
Top item
Double click interval
Font

Callbacks

Browse
Default
Extended
Multiple
Single

Settings

Automatic selection
Selection policy
Size policy
Scroll bar display

Items

Item

The List widget is used to display a list of text items, one or more of which can be selected, depending on the setting of the “Selection policy” resource.

For a scrolling list of text items, use a ScrolledList widget, a composite widget that contains a List widget.

The Items page of the List resource panel lets you add items to the list so you can see what the list looks like. To add an item, enter its text into the “Item” resource box and select “Add”. To remove an item from the list, enter its text into the “Item” resource box and select “Remove”. While items must be added in the order in which you want them to appear, they can be deleted in any order.

To see additional items, change the “Visible items” resource.

The Motif toolkit provides a large number of functions for manipulating Lists such as adding, removing and replacing items. For further details, see the Motif documentation.

Note that each item in a List is a compound string (XmString). It is therefore theoretically possible to use different fonts for different items, or for different parts of a single item. In practice, limitations of the Motif toolkit make this inadvisable.

MainWindow

Scrolled window margins

Width
Height
Spacing

Callbacks

Traverse obscured

Main window margins

Width
Height

Settings

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command location
Message window

The MainWindow provides a standard layout for an application's primary window. This standard layout includes, from top to bottom:

- A menu bar
- A command area with history, a prompt and an input area
- A work area
- A message area

The MainWindow is a composite widget with three Separators and two ScrollBars. You must add the widgets for each element in the standard layout. Use a MenuBar for the menu bar and a Command for the command area. A Text or TextField is usually used for the message area. You must give the message area widget a variable name and specify that name as the "Message window" resource of the MainWindow.

The work area can be almost any other kind of widget. It can be a container widget with other widgets as children. A MainWindow ordinarily displays a scrolled window onto a work area whose size is fixed. If your work area is a Form, you may want to change the "Scrolling policy" resource to "Application defined". This removes the scroll bars and lets the Form resize with the window so that you can use the features of the Layout Editor to control resize behavior.

Note – "Scrolling policy" does not take effect in the dynamic display but works correctly in the generated code.

If you do not add a work area to a MainWindow, the generated code produces warning messages when you run it.

Careful use of resources can make a Form emulate the behavior of a MainWindow. Experience has shown that it is often more convenient to use.

Menu

Display

Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings

Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled[1]
Radio always one
Radio behavior
Resize height
Resize width
Tear-off modal

[1] Only if used as a PopupMenu. It is insensitive in all other cases.

Keyboard

Accelerator [1]
Menu post[1]
Mnemonic
Mnemonic charset

Callbacks

Map
Unmap
Entry

[1] Only if used as a PopupMenu. It is insensitive in all other cases.

The Menu widget provides pulldown, pullright and popup menus and is a specially configured RowColumn widget.

The active items in a menu can be PushButtons, ToggleButtons, or CascadeButtons. Menus can also contain Separators and Labels for display purposes.

To create a pulldown menu, add a Menu as a child of a CascadeButton that is a child of a MenuBar or OptionMenu. When a user clicks on the CascadeButton, the menu appears.

To create a pullright menu, add a Menu as a child of a CascadeButton that is a child of a Menu. When a user clicks on the CascadeButton, the menu appears. Pullright menus are only permitted in menus that are pulled down from a MenuBar, not in OptionMenus.

To create a popup menu, add a Menu as a child of a DrawingArea. When a user clicks on the DrawingArea with the right mouse button, the menu appears. A DrawingArea can have more than one popup menu as a child. In this case, the menu that pops up in the dynamic display depends on which menu is selected in the design hierarchy.

To create a Tear-off menu, set the Tear-off modal resource to enabled. Note that some Motif versions have a bug where, if this resource is not hard-coded and is part of the applications resource file, a call to *XmRepTypeInstallTearOffModalConverter()* must be made from either the main program or the Menu's pre-create prelude for the resource to take effect.

Figure 27-5 shows design hierarchies for the three types of menus.

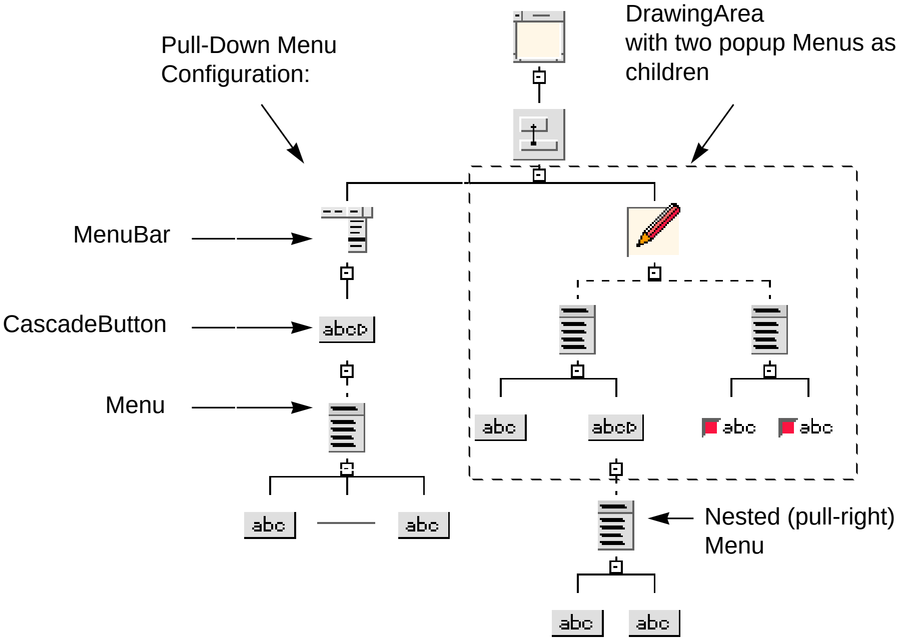


FIGURE 27-5 Sample Hierarchy Using Menus

Unlike pulldown and pullright menus, popup menus must be explicitly managed by the generated code. Sun WorkShop Visual does not do this automatically because popup menus are context-sensitive in most applications. You can do this by using the input callback of the DrawingArea to position and manage the menu, or with an action routine using the translations mechanism. Normally, the menu is positioned using *XmMenuPosition()* and managed using *XtManageChild()*.

To create a Menu with mutually exclusive toggle buttons, set the "Radio behavior" resource to "Yes".

In Motif, Menus are technically siblings, not children, of the DrawingAreas or CascadeButtons from which they appear. However, Sun WorkShop Visual displays its hierarchy as if the Menus were children of these widgets because the DrawingArea or CascadeButton affects the Menu's behavior just as a parent widget does. Sun WorkShop Visual uses a dotted rather than a solid line to connect the Menu to its CascadeButton or DrawingArea. The dotted line indicates that the connection is not a true Motif parent-child relationship.

MenuBar

Display

Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings

Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard

Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks

Map
Unmap
Entry

The MenuBar widget displays a set of CascadeButtons from which you can pull down menus.

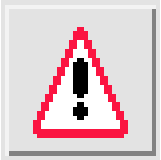
MainWindow, DialogTemplate and SelectionBox provide standard layouts that can include a MenuBar. If you do not use one of these to contain the MenuBar, you must use a Form and attach the MenuBar to its top, left and right sides. For further information about the differences, see the descriptions of the MainWindow, DialogTemplate, SelectionBox and Form. For a design hierarchy that includes a MenuBar with a typical configuration of children, see Figure 27-5 on page 759.

The default resource settings provide a standard menu bar as defined in the *Motif Style Guide*. You can change the “Packing” resource setting from “Tight” to “Column”. “Tight” makes all buttons the minimum size to accommodate their text “Column” makes all buttons the same size

If you use “Column” packing, the “Alignment” resource can be set to center the labels on the buttons. Changing other resources is not recommended.

A MenuBar positions all its CascadeButtons close together starting at the left. If your menu bar has a “Help” button, the *Motif Style Guide* recommends placing it at the right end of the menu bar. To designate a CascadeButton as the “Help” button, enter its variable name as the “Help widget” resource of the MenuBar.

MessageBox



Display	Settings	Callbacks
Message text	Default button	Cancel
Ok label	Dialog type	**Ok**
Cancel label	Alignment	
Help label	Minimize buttons	
Symbol pixmap		

The MessageBox widget displays a message to the user. Sun WorkShop Visual’s error messages are examples of MessageBoxes. The MessageBox is a composite widget that consists of three PushButton gadgets, two Labels and a Separator. These components are contained in a BulletinBoard that is not visible in the design hierarchy. To view the inherited BulletinBoard resources, click on “Bulletin Board Resources” in the resource panel.

Although a MessageBox can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to alert the user.

To display a message or pixmap in the message area, or to change the labels of buttons, change the resources in the resource panel of the MessageBox, not in the resource panels of the component widgets.

You can add a MenuBar and any number of button widgets as children of a MessageBox, as well as a single widget of another type, which becomes the work area. The work area can be a container widget, such as a Form, with children.

OptionMenu

Display

Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings

Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard

Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks

Map
Unmap
Entry

The OptionMenu widget is used to display a one-of-many choice without using the screen space required by a set of radio buttons. The page selectors in Sun WorkShop Visual's resource panels are examples of OptionMenus.

An OptionMenu is a composite widget that includes a Label and a CascadeButton. You should add a Menu child to the CascadeButton, with a PushButton for each choice. You can use Separators to divide groups of options. Figure 27-6 shows a sample hierarchy. Note that you cannot have a cascading option menu.

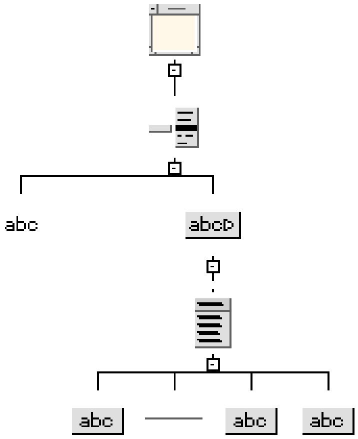


FIGURE 27-6 Sample Hierarchy for the OptionMenu

Set the label identifying the OptionMenu by changing the “Label” resource in the resource panel of the Label. Do not change the label of the CascadeButton as this displays the current setting of the OptionMenu.

PanedWindow

Margins

Margin width
Margin height
Sash width
Sash height
Sash indent
Sash shadow
Spacing

Settings

Refigure
Separator

The PanedWindow widget is used to lay out a set of widgets in a vertical column of uniform width. Each child widget is laid out in a vertical partition that is separated from adjacent children by a movable separator like a window sash. The user can move the sash to determine how much vertical space is allotted to each child. Since the height of a PanedWindow is less than the aggregate height of its children, a PanedWindow saves vertical space without sacrificing functionality. The children of a PanedWindow can be container widgets that control the layout of other widgets.

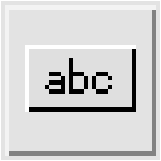
The PanedWindow is a constraint widget. The Constraints panel applies to any child of the PanedWindow, not to the PanedWindow itself. You can display the Constraints panel by selecting “Constraints” from the Widget menu when one of the PanedWindow’s children is selected. The *Resource Panels* chapter discusses how to use this panel.

The Constraints panel lets you set the “Minimum” and “Maximum” height resources for the child. These provide limits on the height of the widget’s partition and positioning of the sashes.

The children in a PanedWindow are constrained to be the same width as the widest child.

You may need to reset a PanedWindow whenever you rearrange or resize its children.

PushButton



Display

Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Margins

Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Settings

Alignment
Type
Resize
Push button
Shadow
Fill on Arm
Fill on select
Indicator as
Indicator type
Multi click
Set
Visible when off

Callbacks

Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Toggles

Widget
Gadget

Keyboard

Accelerator[1]
Accelerator text [1]
Mnemonic [1]
Mnemonic charset[1]
Mapping delay

[1] Sensitive when PushButton is child of Menu

The PushButton widget displays a button that can be “pressed” by clicking a mouse button over it. Like the Label, it can display either text or a pixmap.

There are different kinds of buttons for different needs, such as the ArrowButton and DrawnButton. For a button that pops up a menu, use a CascadeButton.

Setting the “Show as default” resource is not recommended since a BulletinBoard parent often changes this setting. The BulletinBoard decides which button to make the default.

RadioBox

Display

Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings

Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard

Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks

Map
Unmap
Entry

A RadioBox widget is used to contain a group of ToggleButtons that act as *radio buttons*, meaning that they are mutually exclusive. Selecting one toggle in the group deselects the previously selected one. You can set the “Packing”, “Columns”, and “Orientation” resources to create multiple columns as for RowColumn.

The ToggleButtons in the RadioBox are gadgets.

You can make a RowColumn act like a RadioBox by setting its “Radio behavior” resource to “Yes”. This configuration of the RowColumn provides more flexibility than the RadioBox does, e.g. to have Labels, Separators, or other widgets inside the box with the ToggleButtons, or to use the widget version of the ToggleButton instead of the gadget.

RowColumn

Display

Entry border
Margin width
Margin height
Columns
Spacing
Help widget
Last selected

Settings

Orientation
Packing
Alignment
Adjust last
Adjust margin
Aligned
Homogeneous
Popup enabled
Radio always one
Radio behavior
Resize height
Resize width
Tear off modal

Keyboard

Accelerator
Menu post
Mnemonic
Mnemonic charset

Callbacks

Map
Unmap
Entry

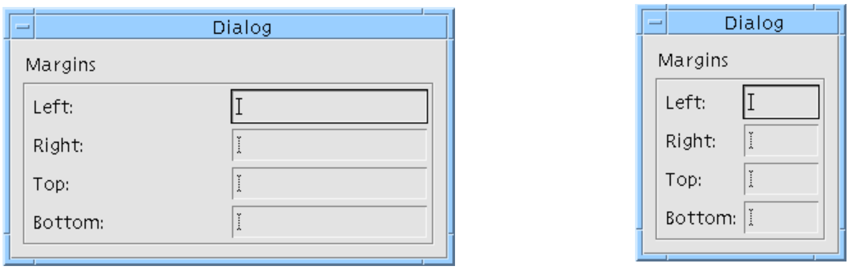
The RowColumn widget is used to arrange child widgets in a grid. It is often used for arranging items such as groups of buttons or toggles. For example, a Menu is a specially configured RowColumn widget. Other widgets that are based on RowColumn are OptionMenu, MenuBar, Menu and RadioBox.

A RowColumn can have any number of children. The default arrangement of RowColumn items is one vertical column. To create multiple columns, set the “Packing” resource to “Column”, then set the “Columns” resource.

Items are read in order starting down the first column when the “Orientation” resource setting is “Vertical” and across the first row when the “Orientation” resource setting is “Horizontal”. When the “Orientation” resource setting is “Horizontal”, the “Columns” setting refers to the number of horizontal rows.

Because a RowColumn widget is not designed to have its layout changed dynamically, it may not display the changes you expect. If its children seem to be the wrong size on the dynamic display, try resetting the RowColumn.

Note – When you use multiple columns, a RowColumn forces all items to be the same width. Sometimes this results in wasted space, as in the “Before” view of Figure 27-7, where the left column has short Labels and the right column has long TextFields. You can resize the TextFields to match the width of the Labels. However, although the new value is accepted in the resource panel, the difference in width is not apparent in the dynamic display until you have changed the value for all of the TextFields, as shown in the “After” view.



Before

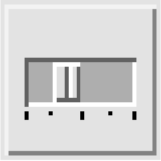
After

FIGURE 27-7 Resizing Widgets in a RowColumn

To create columns of unequal width, use a Form instead of a RowColumn. You can also nest RowColumns to create layouts that are more complex than rows and columns.

To create a group of radio buttons inside a RowColumn, use ToggleButtons and set the “Radio behavior” resource of the RowColumn to “Yes”.

Scale



Display

Decimal points
Minimum
Maximum
Value
Title
Scale width
Scale height
Scale multiple
Font

Settings

Orientation
Direction
Show value

Callbacks

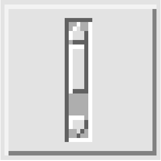
Drag
Value changed

The Scale widget offers a range of values to choose from and displays a slider that can be moved to change the current value. You can drag the slider to move it continuously, click in the trough with the left mouse button to move the slider incrementally, or click in the trough with the middle mouse button to move the slider to the cursor location.

A Scale can have children of almost any type. These are usually Labels, which the Scale lays out evenly along its length.

Changing the orientation of a Scale can have strange effects. If problems occur, try resetting the Scale or its parent.

ScrollBar



Display

Slider size
Minimum
Maximum
Value
Increment
Page increment
Initial delay
Repeat delay
Trough color

Settings

Orientation
Direction
Show arrows

Callbacks

Decrement
Drag
Increment
Page decrement
Page increment
To bottom
To top
Value changed

The ScrollBar widget lets users view data that requires more space than the display area provides. ScrollBars are rarely used alone. It is easiest to use them as part of a composite widget such as a ScrolledWindow, ScrolledList, or ScrolledText.

Each ScrollBar is represented as a rectangle with an arrow pointing outward at each end and a slider inside it. The display area is scrolled either by moving the slider or by clicking on an arrow. You can drag the slider to move it continuously, click in the trough or on the arrows with the left mouse button to move the slider incrementally, or click in the trough with the middle mouse button to move the slider to the cursor location. You can edit the resources to control the amount by which the display area scrolls on each scrolling action.

A ScrollBar cannot have children.

ScrolledList

Margins

Width
Height
Spacing

Callbacks
Traverse obscured

Settings

Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command window
Message window

The ScrolledList widget is a composite widget that displays a scrollable list of items. A ScrolledList is a specially configured ScrolledWindow that contains a List widget. The resources of a List widget child can be set in the normal way.

A ScrolledList resizes itself whenever you add or delete items from the List so that its width always matches that of the widest item in the list. In some versions of the Motif toolkit, the ScrolledList may become confused about its correct width.

To prevent unwanted resizing, you must constrain a ScrolledList in some way. You can constrain it in a Form by using attachments and positions. However, if the Form also contains other widgets, this can produce strange results. To avoid this, use a ScrolledList in a Form containing nothing except a ScrolledList, as shown in Figure 27-8:

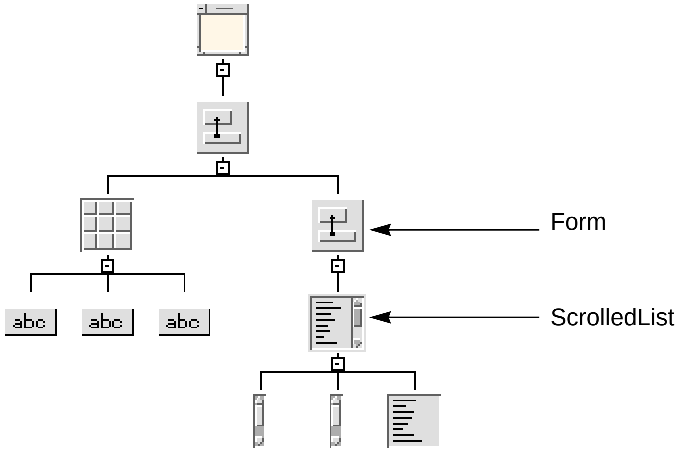


FIGURE 27-8 Effective ScrolledList Placement

You can then place this Form in another Form with other widgets. Attach the ScrolledList to its parent Form on all four sides and set the “Resize policy” of the Form to either “Grow” or “None”. You can set the width and height of the Form to define a reasonable size for the ScrolledList, or fix the initial size of the Form, and therefore the ScrolledList it contains, by using attachments.

Constraints set in the Form supersede the ScrolledList’s “Visible items” resource setting and the width of individual items in the list.

ScrolledText

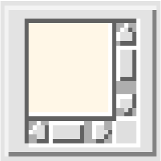
Margins
Width
Height
Spacing

Callbacks
Traverse obscured

Settings
Scroll bar display
Scroll bar placement
Scrolling policy
Visual policy
Show separators
Command window
Message window

The ScrolledText widget is a composite widget that provides a scrollable text area. A ScrolledText is a specially configured ScrolledWindow that contains a Text widget. The resources of the Text widget child can be set in the usual way.

ScrolledWindow



Margins	**Settings**
Width	*Scroll bar display*
Height	*Scroll bar placement*
Spacing	*Scrolling policy*
	Visual policy
Callbacks	*Show separators*
Traverse obscured	*Command window*
	Message window

The ScrolledWindow widget is used to display data that requires more space than is available. It is a composite widget consisting of two scroll bars and a viewing area onto a visible object that can be larger than the ScrolledWindow. A ScrolledWindow can have one child of almost any type.

Although the visible object can be any kind of widget, it is commonly a DrawingArea or a composite widget containing other widgets. For example, a ScrolledWindow can be used to scroll through a form or table of widgets by placing a Form or RowColumn in it. For a scrollable list or text display, use the ScrolledList or ScrolledText widget.

If you do not add a child to a ScrolledWindow, the generated code produces warning messages when you run it.

If the “Scrolling policy” resource is set to “Automatic”, the toolkit handles scrolling for you and the scroll bars are created automatically.

If the “Scrolling policy” resource is set to “Application defined”, you must respond to movements of the scroll bars by changing the information displayed in the ScrolledWindow’s child. In this case, Sun WorkShop Visual generates code to create the scroll bars for you if any resource, callback, or name is set.

The effect of the resources that control scroll bar behavior - “Scrolling policy” and “Scroll bar display” - is not reflected in the dynamic display but they work correctly in the generated code.

SelectionBox

Display

No match string
Pattern
Max history items
Text String
Visible item count
Text columns
Directory mask
Directory

Labels

Apply label
Ok label
Cancel label
Help label
List label
Selection label
Filter label
Directory label

Settings

Dialog type
Minimize buttons
Must match
File type
Work area placement

Callbacks

Apply
Cancel
Ok
No match
Command changed
Command entered

The SelectionBox widget is a composite widget used to select one or more items from a scrollable list. The SelectionBox combination includes a ScrolledList for the item list, two Labels, a Separator and four PushButtons, which are gadgets. These components are contained in a BulletinBoard widget that is not visible in the design hierarchy. To view resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

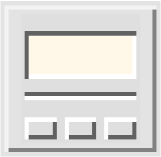
While a SelectionBox can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to get a selection from the user.

To change the labels of button or label widgets, change the resources in the resource panel of the SelectionBox, not in the resource panels of the individual widgets.

You can add multiple children to a SelectionBox. The first child becomes the work area. This can be a container widget containing additional widgets. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the SelectionBox widget’s hierarchy. The additional children can include a MenuBar and any number of PushButton widgets.

The four PushButtons provided are labeled “OK”, “Apply”, “Cancel”, and “Help”. The “Apply” PushButton can be displayed by setting the “Managed” toggle in the PushButton’s Core resource panel.

SelectionPrompt



Display

No match string
Pattern
Max history items
Text String
*Visible item coun*t
Text columns
Directory mask
Directory

Settings

Dialog type
Minimize buttons
Must match
File type
Work area placement

Labels

Apply label
OK label
Cancel label
Help label
List label
Selection label
Filter label
Directory label

Callbacks

Apply
Cancel
OK
No match
Command changed
Command entered

The SelectionPrompt widget is used to prompt the user for text input. It is a composite widget consisting of a Label used for a question or prompt, a Text box into which the answer is typed and three PushButtons (“OK”, “Cancel”, and “Help”). An “Apply” PushButton is also provided. It is displayed by setting the “Managed” toggle in that PushButton’s Core resource panel. These components are contained in a BulletinBoard that is not visible in the design hierarchy. To view the resources inherited from the BulletinBoard, click on “Bulletin Board Resources” in the resource panel.

Most of the information about the SelectionBox applies to the SelectionPrompt, except that the SelectionPrompt does not include a List. While a SelectionPrompt can be used anywhere that a BulletinBoard can be used, it is usually placed in a Dialog Shell that is popped up to query the user for input. To change the prompt or the labels of the buttons, change the resources in resource panel of the SelectionPrompt, not in the resource panels of the individual widgets. The SelectionPrompt can have multiple lines.

The PushButtons in a SelectionPrompt are gadgets. You can add multiple children to a Prompt. The first child becomes the work area. This can be a container widget. The “Work area placement” resource controls where the work area appears in the dialog, even though it appears at the end of the Prompt widget’s hierarchy. The additional children can include a MenuBar and any number of PushButtons.

Separator

Margins	Toggles
Type	Widget
Orientation	Gadget

The Separator widget is a line used to separate objects visually. A Separator cannot have children. Set the "Orientation" resource to specify a vertical or horizontal line. Set the "Type" resource to specify a different line type such as a double line or a dashed line.

Separators can be used to separate items in a Menu or RowColumn or to separate widgets in a dialog box. To separate widgets in a Form, make a Separator a child of the Form along with the other widgets. The Separator is very small until it is constrained in some way. To stretch it the length or width of the Form, attach it to both sides of the Form, or to other widgets on each side. Setting the size of a Separator explicitly is not recommended. A Separator with a "Type" of "No line" can be used as an invisible widget.

Separators are often used inside Menus to divide items into groups. The Separator appears between its adjacent siblings, as shown in Figure 27-9.

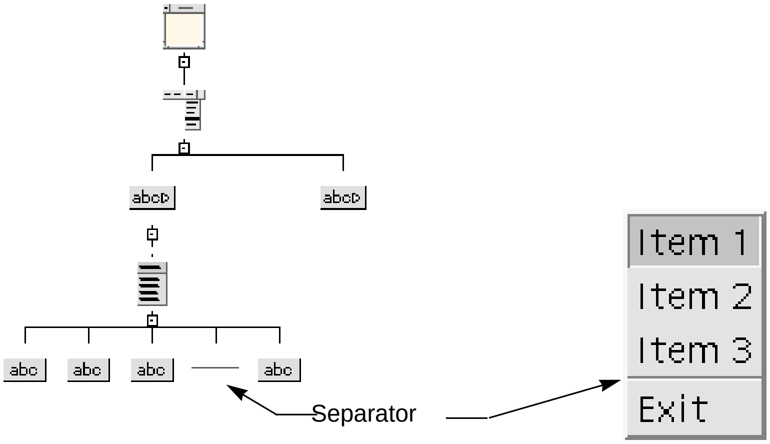


FIGURE 27-9 Use of Separator Inside a Menu

You can use Separators inside a RowColumn. Figure 27-10 shows a sample hierarchy and the resulting dynamic display. When you use Separators in a RowColumn, set the orientation of the Separators explicitly to "Vertical" or "Horizontal". Separators in a RowColumn span a cell the size of every other element in the array. This can produce more white space around the Separator than is pleasing. If you want different proportions, use a Form for your column layout.

Set the RowColumn's "Spacing" resource to 0 to eliminate a gap between adjacent separators.

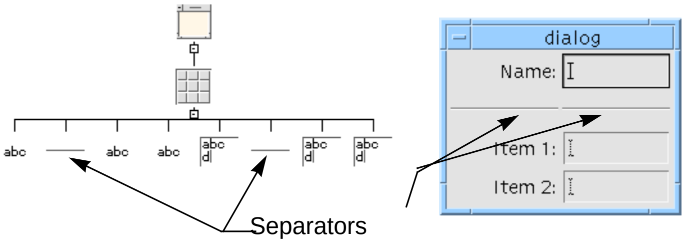


FIGURE 27-10 Use of Separators in a RowColumn (Horizontal Orientation, 4 Rows)

Shells- Dialog, Top Level, and Application

Display

Title
Mwm menu
Icon mask
Icon pixmap
Icon name[1]
Label font
Button font
Text font
Input method
Pre-edit type

Settings

Delete response
Keyboard focus
Input
Transient
Allow resize
Override redirect: No
Iconic[1]
Unit type
Window gravity
Initial state
Save under
Audible warning

Dimensions

Base width
Base height
Width inc
Height inc
Min width
Min height
Max width
Max height
Min aspect X
Min aspect Y
Max aspect X
Max aspect Y
Timeout

[1] Sensitive if Shell is set to Dialog Shell

Callbacks

Pop down
Pop up

Toggles

Application shell
Top level shell
Dialog shell

The Shell widget forms the interface between your design and the Motif window manager. Every Sun WorkShop Visual design hierarchy must have a Shell as its root widget.

The Sun WorkShop Visual palette contains three types of Shell widget - the Dialog shell, Top Level Shell and Application Shell. These Shells can be switched to any of the others by setting the appropriate toggle in the Shell resource panel.

The Application Shell is used as the main application window. Your application must have at least one (and usually only one) Application Shell. Top level Shells look and act like Application Shells. Typically, they are used for all primary windows in the application except the first. Dialog Shells are used for secondary windows such as pop-up dialogs. If an Application or Top level Shell is closed or iconified, all associated Dialog windows also disappear.

An Application or Top level Shell appears as a Dialog Shell in the dynamic display but the generated code produces the correct type of Shell. To check the icon pixmap, set the “Transient” resource to “No”, then reset the Shell. This produces the full set of decorations, allowing you to iconify the dynamic display window.

A Shell can only have one child, which can be of any type. However, much of the Shell’s behavior is based on the assumption that its child is a BulletinBoard, Form, or similar container widget, since the Shell exercises no geometry management over its descendants. A Shell is not visible until it has a child.

Setting a Shell’s width and height on its Core resource panel does not control the window size. To control initial window size, set the *minimum* width and height resources of the Shell, or set the width and height of the Shell’s child.

To control the initial position of a window, set the “Default position” resource of the Shell’s child to “No”, and set the *x* and *y* resources of the child, not the Shell.

Text

Display

Value
Cursor position
Margin width
Margin height
Maximum length
Top position
Selection threshold
Blink rate
Columns
Rows
Font

Callbacks

Activate
Focus
Losing focus
Gain primary
Lose primary
Modify verify
Motion verify
Value changed

Settings

Edit mode
Auto show cursor
Editable
Pending delete
Cursor visible
Resize height
Resize width
Word wrap
Verify bell
Scroll horizontal
Scroll vertical
Scroll left side
Scroll top side

Toggles

Text
Text Field

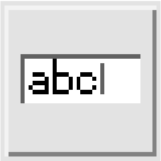
The Text widget provides an area for entering multi-line text. A wide range of callbacks is provided to deal with input verification and validation.

To use multi-line text, you must set the “Edit mode” resource to “Multi line”. To change the height of the Text widget to display multiple lines of text, you can change the “Rows” resource setting to a number greater than 1. Changing the number of Rows may or may not be effective, depending on the type of widget used as the Text widget’s parent.

To create a scrollable text editing area, use a ScrolledText, a composite widget that includes a Text widget. Although the Text widget can be the child of a ScrolledWindow, this configuration does not work well. If you use this configuration, change the “Edit Mode” resource to “Multi line” and increase the number of Rows and Columns to exceed the size of the ScrolledWindow viewing area.

The Motif toolkit provides functions for accessing and modifying the text in the widget. For details, see the Motif documentation.

TextField



Display

Value
Cursor position
Margin width
Margin height
Maximum length
Top position
Selection threshold
Blink rate
Columns
Rows
Font

Settings

Edit mode
Auto show cursor
Editable
Pending delete
Cursor visible
Resize height
Resize width
Word wrap
Verify bell
Scroll horizontal
Scroll vertical
Scroll left side
Scroll top side

Callbacks

Activate
Focus
Losing focus
Gain primary
Lose primary
Modify verify
Motion verify
Value changed

Toggles

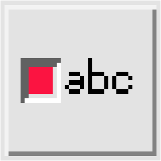
Text
Text Field

The TextField widget is a variant of the Text widget that provides an area for entering only a single line of text. It has all the Text's editing features except multi-line capability.

You can change from TextField to Text by using the toggle. However, to get multi-line capability, you must also set the "Edit mode" resource to "Multi line".

The Motif toolkit provides functions for accessing and modifying the text in the widget. For details, see your Motif documentation. "Books on X and Motif" on page 886 provides some suggestions for further reading on this subject.

ToggleButton



Display

Label
Font
Pixmap
Insensitive pixmap
Cascade pixmap
Arm color
Arm pixmap
Select color
Select pixmap
Select insensitive pixmap

Settings

Alignment
Type
Resize
Push button
Shadow
Fill on arm
Fill on select
Indicator on
Indicator type
Multi click
Set
Visible when off

Callbacks

Activate
Cascading
Arm
Disarm
Expose
Resize
Value changed

Margins

Top
Bottom
Left
Right
Width
Height
Spacing
Default shadow
Indicator size

Keyboard

Accelerator[1]
Accelerator text[1]
Mnemonic[1]
Mnemonic charset[1]
Mapping delay

Toggles

Widget
Gadget

[1] Sensitive when ToggleButton is child of Menu

The ToggleButton widget provides a simple on/off toggle for indicating “yes/no” choices.

ToggleButtons can be made into mutually exclusive radio buttons by placing them inside a RadioBox, or inside a Menu or RowColumn whose “Radio behavior” resource is set to “Yes”. Radio buttons have a different shape from normal toggles, as shown in Figure 27-11.

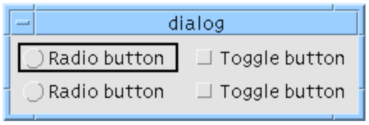


FIGURE 27-11 Radio Buttons and Normal Toggle Buttons

You can configure the ToggleButton to resemble a PushButton that appears to push in and out to represent on and off settings. To do this:

1. **Set the "Shadow Thickness" Core resource to 2. This draws a border around the button.**
2. **Set the "Indicator on" resource to "No". This suppresses the small square indicator.**
3. **Set the left margin to 0. This removes the space which was occupied by the indicator.**

Mapping Motif Widgets to Microsoft Windows

Following is a list of the Motif widgets which can be selected from within Sun WorkShop Visual in Microsoft Windows mode along with the way in which they are mapped to a Microsoft Windows class.

ApplicationShell

Maps to CDialog.

TopLevelShell

Maps to CDialog.

DialogShell

Maps to CDialog.

MainWindow and ScrolledWindow

Map to CWnd unless they are the child of a Shell, in which case they are ignored for Microsoft Windows. If the ScrolledWindow has its Scrolling Policy resource set to "Automatic", it maps to a CScrollView.

Frame, RadioBox and ToggleButton

Map to CButton.

BulletinBoard, Form, RowColumn and DialogTemplate

Map to CWnd if they are structured as a C++ class.

DrawingArea

Maps to CWnd unless its parent is a ScrolledWindow, MainWindow or Shell in which case it is ignored for Microsoft Windows. Otherwise it is forced to be structured as a C++ class.

MenuBar, PopupMenu and CascadeButton

Map to a CMenu and cannot be structured as a C++ class.

OptionMenu

Maps to a CComboBox and cannot be structured as a C++ class.

FileSelectionBox

Maps to a CFileDialog class.

Paned Window

Maps to CSplitterWnd.

Label

Maps to a CStatic.

PushButton

Maps to a CButton if XmNlabelType is XmLABEL or to a CBitmapButton if XmNlabelType is XmPIXMAP.

Separator

This is not mapped to an object on Microsoft Windows - instead it is added as a Menu attribute, if part of a menu. If not in a menu, it is ignored.

Scale and Scrollbar

Map to CScrollbar unless they are part of a ScrolledWindow in which case the appropriate style is added to the enclosing class and they are ignored as widgets. If you choose to “Generate as Resources”, the Scale maps to CSliderCtrl.

TextField and Text

Maps to CEdit.

List

Maps to CListBox.

ScrolledText

This maps to CEdit with appropriate scrolling styles and the ScrolledWindow part is ignored.

ScrolledList

This maps to CListBox with appropriate scrolling styles and the ScrolledWindow part is ignored.

Mapping Motif Resources to Microsoft Windows

Although Microsoft Windows uses resources, the way in which they are used is different from X/Motif. Resources used by Microsoft Windows are compiled into the application. There is also a far more restricted set than on Motif.

Sun WorkShop Visual only generates bitmaps, icons and accelerators as Microsoft Windows resources. Other Motif resources are mapped to visual window attributes or written into the source code.

Window Styles

When a Microsoft Windows object is created, *window styles* can be specified. These are bit flags which are or'd together. The following example shows how a toggle button would be created:

```
Create ( “Classical”, WS_CHILD | WS_VISIBLE | WS_TABSTOP |
BS_AUTORADIOBUTTON, rect, this, IDC_shell_classical);
```

The second parameter to this method, which is a method inherited from a basic MFC class, is the window style. When you set resources in Sun WorkShop Visual, suitable window styles are chosen. Below is a list of the window styles available for each widget which can be mapped to a Microsoft Windows object. The list also shows when they are used and the corresponding Motif resource.

Shells

All Shells have:

- WS_POPUP
- WS_CAPTION
- WS_SYSMENU
- WS_MINIMIZE - if *XmNinitialState* is set to Iconic
- WS_VSCROLL and WS_HSCROLL - if child is MainWindow or ScrolledWindow and the appropriate scrollbar is named, has a resource set or has a callback or method set

ApplicationShell

In addition to Shell styles, has:

- WS_THICKFRAME
- WS_MINIMIZEBOX
- WS_MAXIMIZEBOX

TopLevelShell

Exactly the same styles as ApplicationShell.

Note – This does not mean that ApplicationShell and TopLevelShell are exactly the same on Microsoft Windows - they are different classes.

DialogShell

In addition to Shell styles, has:

- WS_THICKFRAME - unless *XmNoResize* is set to True on the BulletinBoard derived child

MainWindow and ScrolledWindow

- Only supported if *XmNscrollingPolicy* is set to XmAPPLICATION-DEFINED
- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_VSCROLL and WS_HSCROLL - if the appropriate scrollbar is named, has a resource set or has a callback or method set

Frame

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True

- WS_GROUP
- BS_GROUPBOX

BulletinBoard, Form, RowColumn, DrawingArea, and DialogTemplate

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if is *XmNtraversalOn* is True

RadioBox

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_GROUP
- BS_GROUPBOX - if parent is not a Frame

MenuBar, PopupMenu, and CascadeButton

These widgets are not windows on Microsoft Windows, as all other objects are - they are CMenu objects. CMenu is not derived from CWnd. This means that they have no window styles associated with them. Their children (or the children of the PulldownMenu in the case of the CascadeButton) are generated by a call to AppendMenu for each child. The following flags are passed to AppendMenu depending on the type of child:

- MF_POPUP - for a CascadeButton which has a PulldownMenu
- MF_STRING - for CascadeButtons without a PulldownMenu, PushButtons, Labels and ToggleButtons which do not have a valid pixmap object for *XmNlabelPixmap*
- MF_GRAYED - if *XmNsensitive* is False (for the CascadeButton in the case of a MenuBar)
- MF_MENUBREAK - if the item (or the CascadeButton in the case of a MenuBar) starts a new column

The following apply to calls to AppendMenu from a PopupMenu or CascadeButton only:

- MF_DISABLED - for Labels if *XmNsensitive* is True
- MF_SEPARATOR - for separators
- MF_CHECKED - for ToggleButtons which have *XmNset* True

OptionMenu

- WS_CHILD
- CBS_DROPDOWNLIST
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_GROUP - if *XmNnavigationType* is not *XmNONE*
- The SetFont method is called if the widget has a font object resource set for the *XmNbuttonfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell

FileSelectionBox

This maps to a CFileDialog class. Since the Create method is not called explicitly for a CFileDialog (instead InitDialog and DoModal are called) there are no styles. Instead, resources are mapped to parameters passed to the New method:

- OpenFileDialog - always TRUE
- lpszDefExt - always NULL
- lpszFileName - set to the value of *XmNdirSpec* if specified, otherwise NULL
- dwFlags - always OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT
- lpszFilter - if *XmNpattern* is specified this value is set as follows:

```
“<XmNfilterLabelString>(<XmNpattern>)|XmNpattern|All
files(*.*)|*.*||”
```

If *XmNpattern* is not set this parameter is NULL

- pParentWnd - the main window

PanedWindow

- WS_CHILD
- WS_VISIBLE - if the widget is managed

- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- The CreateView method is called to create each of the child panes. This requires that the child pane classes can support dynamic creation (i.e. have the IMPLEMENT_DYNCREATE macro). Sun WorkShop Visual will generate the appropriate macro invocations to support dynamic creation of child pane classes.

Label

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- An alignment (SS_LEFT, SS_CENTER, SS_RIGHT) depending on the alignment of the label (determined either from *XmNalignment* or from parent's *XmNentryAlignment* if parent is a RowColumn)
- SS_ICON - if *XmNlabelType* is set to *XmPIXMAP* and *XmNlabelPixmap* is set to a Pixmap object
- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name.
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource. If the widget is being created (i.e. is not a component) then SetFont will be called if an ancestor BulletinBoard or Shell has *XmNlabelFontList* set.
- The SetIcon method is called if the widget has a valid Pixmap object set for *XmNlabelPixmap.*

PushButton

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- BS_OWNERDRAW - if *XmNlabelType* is set to *XmPIXMAP*
- BS_DEFPUSHBUTTON - if the widget is set as the default button for an ancestor BulletinBoard which is itself a descendant of a DialogShell or a TopLevelShell and there are no CWnd objects intervening between the button and the CDialog

- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource. If the widget is being created (i.e. is not a component) then SetFont will be called if an ancestor BulletinBoard or Shell has *XmNlabelFontList* set.

ToggleButton

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- BS_AUTORADIOBUTTON - if *XmNindicatorType* is set to XmONE_OF_MANY, otherwise BS_AUTOCHECKBOX
- The SetCheck method is called if *XmNset* is set.
- The caption parameter to the Create method is the value of *XmNlabelString* if set, otherwise the widget name.
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource.

Scale and Scrollbar

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False
- WS_TABSTOP - if *XmNtraversalOn* is True
- SBS_HORIZ or SBS_VERT - depending on the setting of *XmNorientation*
- SetScrollRange is called if the widget is a ScrolledWindow component or if either *XmNmaximum* or *XmNminimum* are set.
- SetScrollPos is called if *XmNvalue* is set.

TextField and Text

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_DISABLED - if *XmNsensitive* is False

- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_BORDER - if *XmNshadowThickness* is greater than zero
- WS_GROUP if *XmNnavigationType* is not *XmNONE*
- ES_MULTILINE and ES_WANTRETURN if *XmNeditMode* is set to *XmMULTI_LINE_EDIT*
- ES_READONLY if *XmNeditable* is set to false
- WS_VSCROLL - if the parent is a ScrolledText and *XmNscrollVertical* is the default or set to true
- WS_HSCROLL - if the parent is a ScrolledText and *XmNscrollHorizontal* is the default or set to true
- The SetWindowText method is called if the *XmNvalue* resource is set.
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell.
- The LimitText method is called if the *XmNmaxLength* resource is set.

List

- WS_CHILD
- WS_VISIBLE - if the widget is managed
- WS_GROUP if *XmNnavigationType* is not *XmNONE*
- WS_TABSTOP - if *XmNtraversalOn* is True
- WS_BORDER - if *XmNshadowThickness* is greater than zero
- WS_VSCROLL and WS_HSCROLL - if the parent is a ScrolledList
- LBS_EXTENDEDSEL - if *XmNselectionPolicy* is *XmEXTENDED_SELECT*
- LBS_MULTIPLESEL - if *XmNselectionPolicy* is *XmMULTIPLESELECT*
- LBS_DISABLENOSCROLL - if parent is ScrolledList and *XmNscrollbarDisplayPolicy* is not *XmAS_NEEDED*
- The SetFont method is called if the widget has a font object resource set for the *XmNfontList* resource or if it inherits a font object set for an enclosing BulletinBoard or Shell.

CHAPTER **28**

Troubleshooting in Sun WorkShop Visual

Introduction

This chapter is intended as a quick reference to some common questions and problems new Sun WorkShop Visual users may have. It is organized loosely by functionality:

- Sun WorkShop Visual Interface
- Definitions and Instances
- Unsupported Locale
- Resource Panels
- Layout Editor
- Links
- Code Generation
- Sun WorkShop Visual Replay and Sun WorkShop Visual Capture

The subheadings in this chapter, unlike those elsewhere in this manual, do not describe features of Sun WorkShop Visual but symptoms of problems. Scan the bold sub-headings for a brief description of your problem.

Sun WorkShop Visual Interface

This section discusses problems you may encounter if Sun WorkShop Visual cannot find the correct resource file. Sun WorkShop Visual must be installed so that X looks at the Sun WorkShop Visual resource file. These problems refer to the Sun WorkShop Visual interface, not to the dynamic display.

Labels Don't Display Correctly

Symptom: The labels on the Sun WorkShop Visual buttons, prompts, and menu commands do not display correctly.

Cause and Solution: These labels are only available when the correct resource file is read. If the labels are not available, X substitutes variable names. Reconfigure your system so Sun WorkShop Visual reads the Sun WorkShop Visual resource file. There are several ways to do this in X. Consult your system administrator.

Only a Few Labels Are Wrong

Symptom: Most of the Sun WorkShop Visual display is correct, but a few button labels or other resources are wrong.

Cause and Solution: Your configuration may be reading an obsolete version of the Sun WorkShop Visual resource file. Check the software version and make sure Sun WorkShop Visual is reading the resource file that came with that version.

Blank Help Screens

Symptom: Help screens come up blank.

Cause and Solution: Either your X environment is not finding the correct resource file, or the resource file is not accessing the correct search path. Make sure the "helpDir" resource contains the search path for your Sun WorkShop Visual help database.

Definitions and Instances

This section covers issues connected with the creation and use of definitions and instances. These are described in “Definitions” on page 268.

When opening a file in Sun WorkShop Visual, an error message is displayed claiming a “Class object” is “not recognized”

This means that the design file you are trying to open contains an instance of a definition which is not known to your current Sun WorkShop Visual session. When you create a definition, Sun WorkShop Visual puts a special file, called `.xddefinitionsrc`, into your home directory so that it can find the definition when it is referenced by an instance of it. You will need to find out what the definition is and then do one of the following:

1. Open the design file containing the definition, select the root of the definition, unset the “Definition” toggle in the Widget menu and then select “Define” from the Palette menu. This creates a `.xddefinitionsrc` for you. You do not even need to save the definition design before opening the file containing the instance.
2. Exit Sun WorkShop Visual, copy the `.xddefinitionsrc` from a user who already has the definition defined into your home directory (checking first that you are not overwriting an existing one).
3. Set the `definitionsFileName` resource so that you can share the definitions file with other users, as described in “The Definitions File” on page 269.

Compiling Code Containing an Instance

The following items list some of the problems you may encounter when compiling code which has been generated by Sun WorkShop Visual from designs containing an instance of a definition.

The compiler complains about an “undefined symbol”

Check that the definition and the instance are not in the same design and therefore the same generated code file. The definition must be kept in a separate design. You should then compile the definition code and make it available to the code containing the instance.

The compiler says that it cannot find an include file and then says the instance is "undefined or not a type"

If you have generated the code for the definition into a separate directory, you will have to change the Makefile so that it can find the header file for the definition. This header file should have the same name as that which appears in the Edit Definitions dialog.

The linker says that it has an un "undefined symbol: create_<defintionname>"

The definition code must be compiled in with the code containing the instance. There are two ways of doing this:

1. Compile the code containing the definition into a library and link the library into the application containing an instance of it. You will need to change the Makefile so that the linker can find the library.
2. Generate the code for the definition into the same directory as the code for the instance. Generate a Makefile first with the "New" and "Template" toggles set and then with just the "Template" toggle set. This should create a Makefile which will compile the definition and the instance code into one application.

Unsupported Locale

If you have specified a locale using the LANG environment variable which Sun WorkShop Visual does not recognize, a message is printed on startup informing you that the specified locale is not supported and that it is being coerced to "C". For information on the impact this will have, see "Unsupported Locales" on page 626.

Resource Panels

This section discusses problems you may encounter when you use the resource panels. If you encounter problems with resource values at run time of the generated application, see “Code Generation” on page 804.

For advice on setting resources for specific widgets and combinations of widgets, lookup the widget in Chapter 27, “Widget Reference”, starting on page 743 and your Motif documentation. Note that many apparent problems with resource settings can be solved by resetting the widget involved or the Shell.

When you set values in Sun WorkShop Visual’s resource panels, Sun WorkShop Visual actually resets resources of widget instances in the dynamic display. Sometimes the results are not what you expect:

Resource Settings Are Rejected

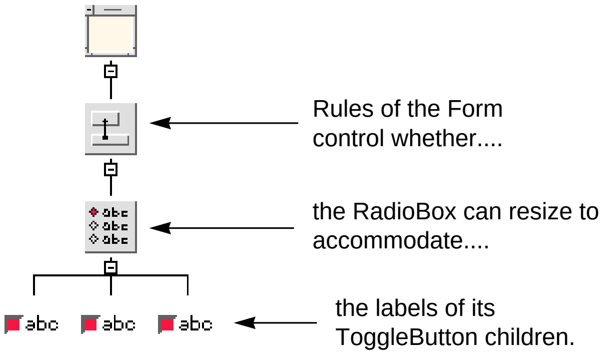
Symptoms:

- The value you typed into a field of the resource panel is not accepted
- The value in the resource panel reverts to the former value
- The value in the resource panel changes to some other value
- The dynamic display reflects the value on the resource panel, but the value is not what you typed

Cause: Motif cannot set the new value you specified. The most common reason is that the selected widget is being constrained by another widget, usually its parent. This is particularly common with size and position resources such as the width and height resources in the Core resource panel, which are frequently overridden.

For example, if a RadioBox contains a group of ToggleButtons, the width of the individual ToggleButtons is determined by their text and margin resources; the width of the RadioBox is calculated from the width of its widest child; and all the ToggleButtons are forced to be the same width as the widest one. A width setting on the Core resource panel is overridden by the calculated value. If the RadioBox is in turn the child of a Form, attachments set in the Layout Editor can override the RadioBox rules.

Solution: Check your design for constraints that may be overriding the setting. Use the constraints imposed by other widgets to achieve the desired effect.



Rules of the Form control whether....

the RadioBox can resize to accommodate....

the labels of its ToggleButton children.

FIGURE 28-1 Resource Relationships

Resource Settings Don’t Take Effect

Symptom: A new value is accepted on the resource panel, but the dynamic display does not immediately reflect the change as you expect.

Solutions: Reset the widget. If that doesn’t work, reset the Shell. If resetting the Shell doesn’t work, consult the *Widget Reference* chapter. You may need to set two or more resources to achieve a single effect. For example, to display a pixmap on a label or button, you must set the “Pixmap” resource to specify the pixmap you want, and set the “Type” resource to “Pixmap.”

A few resources are never reflected in the dynamic display, as discussed below.

Geometry Resources Are Overridden

Symptom: The width and height resources on a widget’s Core resource panel are overridden.

Cause and Solution: The width and height Core resources of any widget can be overridden, either by resources specific to that widget class or by geometry rules of the widget’s parent. Consult Chapter 27, “Widget Reference”, starting on page 743, or Motif documentation for information about resources of the widget and its parent that may be controlling its size.

Cause and Solution: The width and height Core resources of a Shell can be overridden by the corresponding resources of the Shell’s child. One way to control the size of a dialog is to set the width and height resources of the Shell’s child. Another way is to set the *minimum* width and height resources of the Shell, which are not overridden by the child.

Cause and Solution: The *x,y* position resources of a Shell can be overridden by the position resources of the Shell's child if the Shell's child is a BulletinBoard, or a derivative of the BulletinBoard such as a Form, DialogTemplate, or MessageBox. To use the Shell's *x,y* position resources to control the dialog's position, set the BulletinBoard's "Default position" resource to "No."

Resources Are Not Reflected After Resetting

Symptom: The resource panel accepts your resource settings, but they are not reflected in the dynamic display even when you reset the widget.

Cause: Certain resources can only be set when a widget is first created. These resources cannot be changed once the widget is added to the dynamic display. They include various resources related to scrollbars such as "Scrolling policy." Sun WorkShop Visual still lets you change the value on the resource panel, and even though the value is not reflected in the dynamic display, the new setting takes effect at run time of the generated application. For details about individual resources, consult the *Motif Programmer's Manual*.

Cause: The "Dialog Style" resource of BulletinBoards and widget classes that derive from the BulletinBoard can be set to "Modeless," "System Modal," or "Application Modal." Modal dialogs disable all other dialogs until they receive an answer from the user, so if a setting of "System Modal" or "Application Modal" were effective in the dynamic display, you would not be able to do anything else until the dynamic display was closed. You can still select one of these settings; Sun WorkShop Visual disables the setting while you are building the hierarchy and generates it correctly in the code.

Solution: If you have resources of this kind in your design, you can only see the final results after the design is finished and you generate the code. You can see the results at intermediate stages by generating code and running a prototype using the following steps:

1. **Generate a primary code module. Include all types of resources and a** *main()* **program.**
2. **Generate a stubs file (only necessary if your design has callbacks). Leave the function braces empty.**
3. **Compile and link the generated files.**
4. **Run the resulting program.**

Expected Fonts Do Not Display In Sun WorkShop Visual

Symptom: A widget with a default font setting does not display the font you expect in the dynamic display.

Cause: When a widget does not have an explicit font setting, Motif searches back through the design hierarchy to find the widget's nearest BulletinBoard, BulletinBoard derivative, or Shell ancestor, and uses the font setting of that ancestor for the current widget. Therefore, if several buttons all have default font settings but are children of different BulletinBoards, some may show different fonts from others.

Solution: Setting a font on the BulletinBoard or Shell instead of on individual widgets is a convenient way of setting all the fonts at once. To get uniformity you must use the same font on all parent widgets with explicit font settings. A font object is a convenient way to do this.

Font Change on Parent Doesn't Affect Children

Symptom: Changing the font on a BulletinBoard, BulletinBoard derivative, or Shell has no effect on its children.

Cause and Solution: The new font setting on a parent widget does not affect the children until you reset the parent widget. Reset the parent widget.

Cause and Solution: Font settings on parent widgets do not affect their children if the children have explicit font settings of their own. Make sure the children have default font settings.

CascadeButtons in DialogTemplate Don't Display

Symptom: The DialogTemplate does not resize to accommodate CascadeButtons in the MenuBar.

Cause: This is a bug in some versions of Motif. The DialogTemplate resizes properly to accommodate the button box and work area, but if the MenuBar exceeds the width of the button box and work area, it is cut off.

Solution: Add the button box and work area children to the DialogTemplate before you add the CascadeButtons to the MenuBar. In many cases the width of your work area or button box will create enough width to accommodate the MenuBar.

Solution: If your MenuBar is still too wide to display, use a Form as the work area, or put the work area inside a Form, then use the Layout Editor to add extra space around the work area.

Solution: You can force the DialogTemplate to be wider by setting its “Width” resource on the Core resource panel.

Layout Editor

This section discusses problems you may encounter when you use the Layout Editor. Note that many apparent problems in the Layout Editor can be solved by resetting the Form.

Widget Becomes Very Small or Very Large

Symptom: The widget becomes very small or very large.

Cause and Solution: In some versions of Motif there is a bug in the Form that appears when the Form is a child of a DialogTemplate. When you reset the Form, any container widgets inside the Form become very small. If you have this problem, resetting the DialogTemplate instead of the Form corrects it. To do this, you must go from the Layout Editor screen to the main Sun WorkShop Visual screen, select the DialogTemplate in the construction area, and give the “Reset” command. To resume working in the Layout Editor, first select the Form again in the construction area.

Cause and Solution: Attachments can change the size of a widget. For example, attaching the edges of a widget to the corresponding edges of the Form forces the widget to span the full width or height of the Form. Reset the Form. If resetting the Form doesn’t work, remove some of the attachments from the widget and reset again.

Methods of breaking attachments are listed below. For a more complete discussion, see Chapter 4, “The Layout Editor”, starting on page 97.

- Use “Undo” to remove the most recent attachment
- Move the widget
- In the “Attach” mode, click just inside the edge of the widget or drag from just inside the edge of the widget toward the widget’s center
- Replace the old attachment with a new attachment
- Select the widget in the design hierarchy, bring up its Constraints panel, and reset the attachment “Type” for that edge to “None”.

“Circular Dependency” Error Message

Symptom: A “circular dependency” message appears after you make an attachment.

Cause and Solutions: If you add an attachment that results in a circularity involving two or more widgets, Motif detects the circularity and returns the error message. Click on “Undo.” If you still get the error message, carefully inspect your layout for an attachment loop and remove one of the attachments.

“Bailed Out...” Error Message

Symptom: A “bailed out” message appears after you make an attachment.

Cause and Solutions: This Motif message indicates that your layout contains attachments that contradict one another without being circular. It usually occurs with “Self” or “Position” attachments. Use “Undo” or move the widgets to remove the contradictory attachments, then reset the Form.

Widgets Overlap the Boundary of the Form

Symptom: Widgets at the edge of a Form cause breaks in the Form’s boundary line.

Causes: Widgets whose edges coincide with the sides of the Form can overlap the line drawn around the Form, causing undesirable breaks in the line. This only occurs if the Form is the immediate child of a Shell. Three conditions in the Form can cause the overlap:

- A widget is attached to the edge of the Form with an offset of 0 or 1 pixel
- A widget is attached to the edge of the Form with a default offset and a vertical or horizontal spacing value of 0
- There are no attachments between the bottom or right edge of the Form and the widgets closest to those edges

Solution: Attach widgets to the top or left edge of the Form with an offset of 2 or more pixels. You can use an explicit offset, or you can set the Form’s vertical and horizontal spacing resources to the offset value and use the default offsets. Make sure the widgets at the bottom and right side of the layout are attached to the edge of the Form with an offset or spacing of 2 or more pixels.

Solution: Put the Form inside another manager widget, such as another Form or a DialogTemplate. This is the simplest and most flexible solution.

Links

This section discusses problems you may encounter when you use the “Edit links” command in the Widget Menu. For additional information about links, see the *Code Generation* section that follows.

“Add” Is Disabled

Symptom: The “Add” option is grayed out.

Cause and Solution: The Link facility requires the target widget to have an explicit variable name. If the target widget is a Shell, its immediate child must also have an explicit name. Sun WorkShop Visual grays out the “Add” option if it does not find explicit names. Name the appropriate widgets.

A Link Stops Working

Symptom: A link that used to work stops working. The link appears in the “Edit links...” dialog with a blank space instead of an icon.

Cause and Solution: If you change the name of a widget, Sun WorkShop Visual does not automatically update links that refer to that widget and they cease to be functional. Remove the obsolete links and replace them with new ones.

Links Don’t Update When You Select Another Button

Symptom: The “Links” panel doesn’t behave like the resource panels. If you edit links on one button then select another button, the “Links” panel still shows the links from the previously selected button.

Cause: Sun WorkShop Visual interprets the selection of any new widget as a target widget for a potential new link on the previously selected button.

Solution: Pull down the Widget Menu and select “Edit links” again to display and edit links on the second button. You do not have to close the Links panel first.

Code Generation

This section discusses problems you may encounter when you generate code. Some of these problems result because Sun WorkShop Visual offers you so much flexibility in arranging your files. For example, you should make sure to generate Link functions in only one file, and to generate Includes in the files where they will be needed. Read “Arranging Your Files” on page 237 for more details.

“No Application Shell” Warning

Symptom: Sun WorkShop Visual displays a “No Application Shell” warning message when you try to generate the primary module with a *main()* program.

Cause and Solution: Your design does not contain the required Application Shell. Bring up the resource panel for the Shell of the main window in your design, click on the “Application Shell” toggle, then click on “Apply.”

Links are Undefined

Symptom: Link functions are undefined at link time.

Cause and Solution: If you generate Links with your primary module, you must generate the actual code for the links - the Link functions - into one of your code files, either the primary module or a stubs file. Be sure to generate the link functions into an appropriate file.

Global Widgets Are Undefined

Symptom: Global widgets are undefined when you compile the stubs file.

Cause and Solution: Declarations of global widgets and objects are generated into the primary module, but not into the stubs file. To generate a header file that declares them, use the “Externs...” option and *#include* the resulting header file with your callbacks. This is preferable to writing your own *extern* declarations or copying the ones generated in the primary module, because it is less error-prone and more complete. The Externs file can be regenerated when necessary to reflect changes in the design.

Application Does Not Use Resources from X Resource File

Symptom: The generated application doesn't use the resource values from the generated X resource file. For example, variable names appear on labels and buttons instead of the label strings, colors are wrong, or fonts are wrong. The exact symptoms depend on which resources were generated into the X resource file and which were hard-wired.

Cause and Solution: You did not regenerate the X resource file when you regenerated code. If you have added or removed widgets from your design, default widget names may change and no longer correspond to those in the generated X resource file. Regenerate the X resource file.

Cause and Solution: X cannot find the X resource file when you run your application. You may need to rename the X resource file, usually with the same name as the application class and without a suffix. For more information, see your X documentation.

Cause and Solution: X cannot recognize the widget names in the file because a different application class name was used for the generated code file and the X resource file. Regenerate the application and the X resource file, being sure to use the same application class name for both. Be sure to use a unique application name to avoid confusion with other resource files your system may be accessing.

Default Resources Change At Run Time

Symptom: A color, font, or other resource is different at run time from that in the dynamic display.

Cause: Some resources shown in the dynamic display are inherited from Sun WorkShop Visual. If not explicitly set on the resource panels, these resources may inherit values from other sources at run time, depending on the platform where the program is run.

Solution: To ensure the correct colors and fonts, set explicit values for them on the resource panels. Foreground and background colors can be set on each Shell in the design and are then inherited by all children of the Shell. Fonts can be set on BulletinBoards, derivatives of the BulletinBoard, or Shells, and apply to all their children.

Unexpected Results Occur When Widgets Share a Widget Name

Resource values can be shared among widgets with a common widget name, but only if they are read from the X resource file into the resource data base. The following rules apply:

- Sharing of resources occurs only at run time. Resource values are not shared among widgets in the dynamic display
- Hard-wired resources are not shared
- Object bindings are not shared

Symptom: Resource values are different at run time from values in the dynamic display. When a resource is generated to the X resource file, the result is different from when it is hard-wired.

Cause: These are expected results when widgets share a widget name. The dynamic display and hard-wired resource settings disregard common widget names. Resources generated into the X resource file, however, affect all widgets with a common widget name.

Symptom: A color or font is not shared among widgets with a common widget name, even if that resource was generated to the X resource file.

Cause and Solution: The color or font used an object binding instead of a color or font setting. Use a simple color or font setting, or set the resource on a common parent of the widgets that you want to share the value.

Symptom: An explicitly set resource value is overridden at run time.

Cause: If widgets share a widget name and resources are generated to the X resource file, only one value is used even if more than one was set.

Solution: Generally it is better to avoid common widget names unless widgets are to share all resource values. However, you can force any resource value to be restricted to a single widget by hard-wiring it. Use the masking toggles on the resource panels to do this.

Sun WorkShop Visual Replay and Sun WorkShop Visual Capture

The following section answers frequently asked questions about Sun WorkShop Visual Replay and Sun WorkShop Visual Capture.

Why is it not possible to record and replay certain applications (e.g. Netscape)?

Typical reasons are:

- The application may be statically linked with the Xt library rather than dynamically linked with it.
- The application may have its own multi-threading scheme that disallows Xt Work Procedures.
- The product may have multiple application shells. (See below)

If you experience difficulties in recording and replaying your own software, simply relink it so that it uses *libXt.so*.

Why is the click position in a text widget not recorded?

All of the “position sensitive” motif widgets are recorded / replayed through special Sun WorkShop Visual Replay routines. You will find the source code for these routines in the *src/examples/replay/libcvtXm* directory.

The conversion routines for *XmText* and *XmTextField* are not built by default because, for most testing purposes, it is reasonable to treat the text field as a simple data entry field whose contents you wish to replace.

It is a lot simpler to do the following:

```
doubleclick mytextwidget
type halloworld
```

than it is to do this:

```
doubleclick mytextwidget(position, 25)
type halloworld
```

It would also be difficult to check that doubleclicking at a particular character position did select all the text.

Different test runs may involve replacing the contents of a text field with different values. The name of the text widget is the most important item - not the values which are to be placed in it.

If you wish to test the editing facilities provided in an *XmText* widget within an application, you should rebuild the *libcvtXm* directory with *-DHANDLE_TEXT* added to the `cc` command line. Then copy the *libcvtXm.so* shared object to *lib/xds* so overwriting the standard version.

The Sun WorkShop Visual Replay copyright message appears but then it exits

This commonly occurs when Sun WorkShop Visual Replay has determined that you are attempting to replay a non-Motif application.

The `-O` flag can be used to force the application to be invoked and allows you to replay any non-Motif application functionality.

NB: This situation can also arise for Motif applications which have multiple application shells (see below).

Sun WorkShop Visual Replay is invoked successfully but appears not to be working

Typical reasons are:

- The application has a "splash" or creates one or more temporary ApplicationShell widgets on startup.

By default, Sun WorkShop Visual Replay registers the first ApplicationShell widget and uses it as a point of reference. Usually it "takes" the first applicationShell it sees, but you can change it to take the 4th by adding:

```
-use 4
```

on the command line, or set the *XDSUSESHELL* environment variable:

```
setenv XDSUSESHELL 4
```

NB: To check that you are using the "real" application shell, try recording. You should see:

```
in ApplicationShell
      ......
```

If you see:

```
in myapp
      .....
```

then it may not be using the correct application shell. (If your application has an unmapped application shell, and multiple toplevel shells, then this *is* the correct behavior).

The solution is to keep incrementing the `-use` count. The worst that can happen is that you tell it to ignore ALL you application shells, so it won't record, replay or capture.

- Your application has reworked or subverted Xt event handling.

You know that this is the problem, if:

  - the `-i` flag is used and the application appears, but the Sun WorkShop Visual Replay dialog does not, or
  - the `-I` flag is used and the Sun WorkShop Visual Replay dialog is displayed *before* the application appears.

In these circumstances, all you will be able to do is capture designs.

- Your application has not been linked with Motif.

In this situation, the default behavior is for Sun WorkShop Visual Replay to abort. This may be overridden using the `-O` flag. Note however that you will not be able to record or replay any application functionality which relies upon Motif.

My application has 3rd party widgets in it. How can I capture them properly?

The capture mechanism creates capture files in the Sun WorkShop Visual *.xd* format and assumes that you are using the standard version of Sun WorkShop Visual, i.e. not one supplemented with non-Motif widgets. By default, all non-Motif widgets are represented in the capture file as Motif DrawingArea widgets.

If you are using a version of Sun WorkShop Visual which supports the 3rd party widgets you wish to capture, you need to set the following resource:

```
*xdsCaptureUserWidgets:true
```

Can Sun WorkShop Visual Replay handle japanese (and other) text and input methods?

Yes. It records the composed text that has been inserted in the text field and replays by inserting the text directly. It has been configured for the Motif Text and Textfield widgets. The configuration software is in:

`src/examples/replay/motif/motif4.c`

and the mechanism for registering the software is in:

`src/examples/replay/motif/register.c`

The configuration involves a `get/put` routine of the data. Some input methods will allow you to access this information. The default fallback is to access the string in the widget itself.

It has hard-wired control-space as an input method compose request, and has an alternative “compose” keysym resource, which is set by default to Henkan_Mode.

If you are using the recording software with an input method that takes, e.g. F3 as the compose key, you should run your software with:

```
-xrm *xdsImComposeKeySym:F3
```

or set this resource from a defaults file, or with `xrdb`.

Can my customers record/replay my applications?

To permit users to use Sun WorkShop Visual Replay to record and replay your application, you must have the following line in your code:

```
xdsAllowUserAccess()
```

and link the application with the *libxdsclient.a* library.

APPENDIX **A**

Sun WorkShop Visual Replay Command Syntax

Introduction

This appendix describes the keywords used in Sun WorkShop Visual Replay scripts.

The Sun WorkShop Visual Replay script keywords have been divided into the following subsections according to their functions:

- Record and Replay Commands:
  - Specifying the context of actions
  - Button actions (Simple controls)
  - Pulldown Menu operations
  - Option Menu operations
  - Keyboard Operations
  - Text Entry
  - Button actions (Position dependent controls)
- Extra Commands:
  - Resource Evaluation
  - Widget Hierarchy Analysis
  - Non-application operations
  - Condition clauses
  - Display Expressions
  - Widget State Expressions
  - Importing User-Defined Commands

Specifying the Context of Actions

Keywords

in - specify the context of subsequent actions in a script

ApplicationShell - the top level shell of the application

Synopsis

```
in shell_widget
        commands
in ApplicationShell
        commands
```

Inputs

shell_widget	the name of a shell widget other than the main application shell

Description

Sun WorkShop Visual Replay scripts consist of actions on widgets. These actions have to take place within the context of the shell (i.e. dialog) which contains that widget. If the shell is not realized, the script will fail at that point. The *in* command cannot be nested. Once you have come out of a shell (to go into another shell), you must go back *in* to that shell before attempting any further actions within that context.

Examples

```
in ApplicationShell
    push this_button
    push that_button
  push help_dialog_button
in help_dialog_popup
```

```
  push cancel_button
in ApplicationShell
  push another_button
```

Button Actions (Simple Controls)

Keywords

push	- press and release a mouse button
doubleclick	- doubleclick mouse button

Synopsis

```
push widget [with [modifier-]button[1-5]]
doubleclick widget
```

Inputs

widget	the name of a widget
modifier	a keyboard modifier
button[1-5]	the number of the mouse button (default is button 1 with no modifiers).

Description

push simulates a single click (a mouse button press/release sequence) using a mouse button on the named widget. The ***with*** keyword allows you to specify a particular mouse button. If this is not used, button1 (the left mouse button) is used. A keyboard modifier (such as the Shift key) can be used to extend the permutations of mouse button events. The permitted modifiers are alt, ctrl and shift.

doubleclick simulates a doubleclick with the left mouse button. This can be used in any widget but is especially useful for selecting from a text widget (see “Text Entry” on page 819).

Usage

In some widgets, where the user clicks with the mouse is unimportant. For example, clicking on a button widget in any part of it will activate that button. However, for other widgets, the position is significant; for example pushing on a scale widget will have different effects depending upon the where the push was made.

The following table lists those widgets which are position and non-position dependent:

TABLE A-1

Position Independent Widgets	Position Dependent Widgets
Buttons	Sliders
Toggles	Scales
	Lists
	Drawing Areas
	Text Widgets[1]
	Non-Motif widgets

[1] Recording and replaying user interaction with text widgets is covered in "Text Entry" on page 819

Refer to "Button Actions (Position Dependent Controls)" on page 820 for details on recording and replaying the other widgets in the position-dependent list.

Examples

```
in ApplicationShell
  push this_button
in ApplicationShell
  push that_button with shift-button2
in my_dialog_popup
     if color_toggle->set:true
           push color_toggle
     endif
```

Menu Operations

Keywords

cascade	- post a pulldown menu
pullright	- post a pullright menu from a pulldown menu

Synopsis

```
cascade cascadebutton
        select widget
cascade cascadebutton
        pullright cascadebutton
```

Inputs

cascadebutton	the name of a cascadebutton
widget	the name of a widget within the cascade button’s pulldown menu

Description

cascade is a shorthand way of describing menu operations. You can also post a menu by pushing on the associated cascade button or using a keyboard accelerator. Similarly, menu options can be selected using accelerators or keyboard mnemonics.

cascade posts a pulldown menu to allow a selection to be made from it. The selection may be a widget (i.e. an option in that menu) or a cascadebutton which displays a pullright menu.

Examples

```
in ApplicationShell
     cascade file_m
        select open_file
in ApplicationShell
```

```
     cascade format_menu
        pullright character_menu
```

Notes

Sun WorkShop Visual Replay only supports one level of pullright menu to conform to the Motif style guide. You can however use the push command in your scripts to select pullright menus in succeeding levels.

Option Menu Operations

Keywords

Synopsis

option *opmenu-widget*::*member_widget*

Inputs

Description

option selects an option from an option menu.

Examples

```
in ApplicationShell
     cascade format_menu
          pullright character_menu
          option character_menu::bold
```

The next example only selects an option if the option menu itself is sensitive to user input:

```
if IsSensitive(myoptionmenu->OptionButton)
     option myoptionmenu::thisoption
```

```
endif
```

If you want to check the current setting of the optionmenu (i.e. what was last selected), you simply examine the option menu *menuHistory* resource, for example:

```
     if myoptionMenu->menuHistory: select_yes
          message he said yes
     endif
```

Notes

An alternative method of selecting a member of an option menu is to push the option button and then push the appropriate member widget. However, we recommend use of the ***option*** syntax as it more closely mimics user actions.

Keyboard Operations

Keywords

alt	-	select current word
ctrl	-	select current line
key	-	enter a keysym from the keyboard

Synopsis

`alt` *`char`*

`ctrl` *`char`*

`key` *`keysym`*

Inputs

char a single character

keysym any X keysym (see *X11/keysymdef.h* for list)

Description

Keyboard input is directed at the widget that has the focus. Sun WorkShop Visual Replay does not require any extra programming to enter input from the keyboard.

Users and test scripts alike have to work with the window manager when entering text. Where explicit focus is in place (i.e. you have to click in a window to get the focus), you will have to program this into the test script.

Example

```
in ApplicationShell
          alt f
          type o
in open_file_popup
          multiclick selection_field
          type foo.xd
          push ok_button
          doubleclick my_text_field
          type hallo world
          key Return
```

Notes

A push or a doubleclick in a text field has the side effect of taking the focus. This is the only place in Sun WorkShop Visual Replay that focus is handled directly.

Data entry into text fields often overrides what is already there and will be preceded by a doubleclick or a multiclick.

Text Entry

Keywords

type	-	enter text from the keyboard
key	-	enter a keysym from the keyboard
doubleclick	-	select current word
multiclick	-	select current line

Synopsis

```
type text
key keysym
doubleclick textwidget
multiclick textwidget
```

Inputs

keysym	any X keysym (see *X11/keysymdef.h* for list) without the XK_ prefix
textwidget	the name of a text widget
text	a text string

Description

Most text widgets in an application are used for single line data entry (for example the selection fields in a File Selection Box). Sun WorkShop Visual Replay allows testers to replace the default content of the field with a known value and then check the consequences.

type enters text into a text widget. ***doubleclick*** and ***multiclick*** program word and line selection respectively. ***multiclick*** is most commonly used in test scripts, when you want to replace the contents of the text field, regardless of how many words there are on the line.

Examples

```
in form_attr_dialog_popup
          doubleclick formHorizSpacingField
        type 100
in coreDialog
          multiclick title_t
          type My Dialog Title
```

Notes

There is a limit of 512 characters to the length of a line which can be handled by Sun WorkShop Visual Replay. In you want to enter a text string whose length exceeds this limit, split the text and *type* in each section.

Sun WorkShop Visual Replay works around a problem in some versions of Motif where triple-click is not properly handled in XmTextField widgets. In these circumstances, if your script contains ***multiclick***, it will be converted to ***doubleclick***.

Button Actions (Position Dependent Controls)

Keywords

push - press and release a mouse button

drag - combine a press and release within the same widget

Synopsis

```
push widget(mame,qual)
drag widget(name1,qual1)-widget(name2,qual2)
```

Inputs

widget	a widget name
name, name1, name2	application / widget dependent description

Description

In some widgets (e.g. drawing areas) where you click is important. In the case of drawing areas, a position within the drawing area is needed. For lists, you need an indication of which item has been selected. The version of ***push*** listed above is intended for such position-dependent widgets.

In these widgets, you will often need to do more than just click. You may need to press down at one point and release at another. An example is the setting up of attachments between widgets in the Sun WorkShop Visual form layout editor. This may involve a server grab, so it is described as a single ***drag*** operation where the first part describes where you pressed and the second where you released the button.

This mechanism can be used for single user-defined widget instances, such as the drawing areas within your application and also for entire widget classes (as we have done for XmList, XmScale and XmScrollBar and various 3rd party widget sets).

Example

The first example shows how the Motif DrawingArea widget has been implemented for Sun WorkShop Visual testing:

```
in ApplicationShell
  push tree_da(mybutton,centre)
```

In the next example we show how attachments are made between the *frame1* and *button_box* widgets in the Sun WorkShop Visual form layout editor:

```
in form_layout
     drag layout(frame1,right)-layout(button_box,left)
```

You can try out these effects in Sun WorkShop Visual.

Notes

Information on how to handle your own position-dependent widgets, or those from a 3rd party supplier, are given in “Extending the Sun WorkShop Visual Replay Widget Set” on page 458.

Resource Evaluation

Keywords

printres - print the value of a widget resource

Synopsis

`printres` *`widget->resource`*

Inputs

widget	the name of a widget
resource	the name of the widget resource

Description

printres prints the current value of a specified resource within a selected widget. This is especially useful in test scripts where a known resource value is expected. The name of the resource must be specified without any “XmN” prefix, e.g. “labelString”.

Your scripts are more likely to include resource evaluation within conditional expressions.

Example

```
in my_shell
    if !my_option_menu->menuHistory:default_option
         message FAIL: bad setting for my_option_menu
         message Setting should be:
         printres my_option_menu->menuHistory:default_option
    endif
```

Widget Hierarchy Analysis

Keywords

tree - produce recursive listing of current widget hierarchy

dump - show resources assigned to widget

snapshot - produce recursive listing of current widget hierarchy and the resources assigned to each widget

Synopsis

```
tree widget
dump widget
snapshot widget
```

Inputs

widget the name of a widget

Description

The *tree*, *dump* and *snapshot* commands allow you to analyze the structure of the widgets within an application interface and the values of resources assigned to those widgets. The results from the analysis are displayed on standard error.

tree gives a recursive listing of widget names in the widget hierarchy from the nominated widget.

dump displays the resource settings of the nominated widget.

snapshot displays the resource settings of the nominated widget and all other widgets in the widget hierarchy from the nominated widget.

Example

The following command displays the resources allocated to the button1 widget:

```
in ApplicationShell
    dump button1
```

Part of the example output is shown below:

```
button1():
    Boolean ancestorSensitive:true
    HorizontalDimension width:58
    VerticalDimension height:22
    Pixel background:color('black')
    Pixel foreground:color('#72729F9FFFFF')
    HorizontalDimension highlightThickness:1
    Pixel highlightColor:color('black')
    XmString labelString:'Button A'
  Pixel armColor:color('red')
```

The next command displays the widget hierarchy from the form1 widget:

```
in ApplicationShell
    tree form1
```

Part of the example output is shown below:

```
    rowcol1():
         buttonA():
         button2():
    address_area():
         label1():
         text1():
```

Notes

Sun WorkShop Visual Replay assigns a unique name to widgets which share a common widget name within a shell (e.g., HorScrollBar#1, HorScrollBar#2, Apply#3, Apply#5, etc.). Where the replay name is different from the actual widget name, it is given within the brackets.

Non-Application Operations

Keywords

delay	-	pause replay of user actions
message	-	print message
sequence	-	label part of a script
shell	-	execute shell command

Synopsis

```
delay duration
message text
sequence text
shell command
setenv env-var env-value
breakpoint widget
exit status
```

Inputs

duration	time in seconds
text	a text string
widget	the name of a widget
status	either 1 or 0

Description

delay allows you to insert a pause in a script. This is useful when you wish to visually inspect the application at particular points in its execution. The next action in the script will continue after the pause.

message displays a message on standard error. This allows you to label different parts of the script and communicate expected results and errors to testers. The message text does not have to be enclosed in quotes.

sequence is used to label different sections of a script. Then if an error occurs, you can skip to the next labelled sequence and continue from that point.

To use ***sequence***, you must invoke `visu_replay` with the *-skip-on-error* flag. By default, `visu_replay` is run with the *-user-on-error* flag which will stop the test and stay in the application when an error occurs. The remaining error flag, *-exit-on-error* causes will terminate the application when an error occurs.

shell executes a shell command from a script. The script continues when the shell command has terminated. This facility allows you to enrich your scripts to do far more than simply re-running user actions.

setenv is used in conjunction with the ***shell*** command to pass information to the shell through environment variables. ***setenv*** has two arguments. The first is the name of the variable; the second is an expression that can combine widget resource values and one of the following convenience functions:

- `WindowId(widget)`
- `WindowFrame(widget)`
- `Parent(widget)`
- `Shell(widget)`

breakpoint is used, in conjunction with a debugger, to set a breakpoint in a script when a nominated widget is activated. You can then examine the internals of individual widgets.

A script which contains the ***breakpoint*** keyword should be invoked as follows:

```
visu_replay -f script debugger app
```

where *script* is the name of the script, *debugger* is the name of your debugger and *app* is the name of the application to be exercised by the script. The debugger is run by Sun WorkShop Visual Replay. At the ***breakpoint*** keyword, the application will stop as if you set the breakpoint directly. This will allow you to inspect widget internals even if your application has been optimized.

exit terminates the script with the specified exit status.

Examples

To delay for 5 seconds after pushing a widget:

```
in ApplicationShell
    push mywidget
    delay 5
    push yourwidget
```

To take a screen dump of a shell without window manager decorations:

```
in ApplicationShell
```

```
          setenv ID WindowId(ApplicationShell)
          shell xwd -id $ID -out /tmp/shell.xwd
```

To take a screen dump with window manager decorations:

```
     in ApplicationShell
          setenv ID WindowFrame(ApplicationShell)
          shell xwd -id $ID -out /tmp/shell.xwd
```

To take a screen dump of a pulldown menu, when you only know the name of its cascade button:

```
     in ApplicationShell
          push cascade_button
          setenv ID WindowId(cascade_button->subMenuId)
          shell xwd -id $ID -out /tmp/shell.xwd
```

Note – If you don't push the button first, the menu will not have been posted and xwd will not be able to snapshot it.

To do the same with an OptionMenu:

```
     in ApplicationShell
          push option_menu.OptionButton
          setenv ID WindowId(option_menu->subMenuId)
          shell xwd -id $ID -out /tmp/shell.xwd
```

To note the background color of the cascade button's parent:

```
     in ApplicationShell
          setenv ID Parent(cascade_button)->background
          shell echo The Color $ID
```

Condition Clauses

Keywords

```
if
else
elif
```

```
endif
```

Synopsis

```
if expression
    actions
[elif expression
    actions]
[else
    actions]
endif
```

Inputs

expression	an expression which evaluates to true or false
actions	one or more user actions

Description

The ***if*** statement allows the control flow through a script to be sensitive to conditions inside the application as it is being run. For each ***if*** there must be a matching ***endif***. If necessary the statement can include optional alternatives (***elif***) and a default catch-all ***else*** condition.

Example

```
in my_shell
    if !my_option_menu->menuHistory:default_option
        message FAIL: bad setting for my_option_menu
        message Setting should be:
        printres my_option_menu->menuHistory:default_option
    else
        message setting ok for my_option_menu
endif
```

Display Expressions

Keywords

```
IsPseudoColor
IsDirectColor
IsTrueColor
IsStaticColor
IsStaticGrey
IsGreyScale
```

Synopsis

```
if expression
    actions
endif
```

Inputs

expression one of the keywords listed above

Description

You cannot guarantee that a script recorded on one display will necessarily work on another of a different type. Certain applications make heavy use of color and may display a color restriction message to a user if he is running the application on a display with a limited color map. Your scripts must accommodate such situations.

Example

```
if !IsPseudoColor
    message Non PseudoColor display
    in warning_popup
         push warning.OK
endif
```

Widget State Expressions

Keywords

```
IsVisible
IsManaged
IsRealized
IsHere
```

Synopsis

```
if expression
    actions
endif
```

Inputs

expression one of the keywords listed above

Description

Where parts of a dialog are selectively displayed, you can check which parts are managed and realized using the ***IsManaged*** and ***IsRealized*** expressions.

IsVisible is intended for small (VGA) displays where the whole of a dialog may not be visible on the screen. This is important as Motif TAB navigation traversal model ignores controls which are off screen.

IsHere simply checks whether the widget exists in the current shell.

Example

```
in ApplicationShell
    cascade file_menu
        select fm_menu.fm_exit
    if IsVisible(save_dialog)
        in save_dialog
```

```
                push save.ok
else
      message Save Dialog cannot be seen
endif
```

Importing User-Defined Commands

Keywords

import	- load a module of additional commands
user	- invoke a command from a loaded module

Synopsis

import *module*

user *command text*

Inputs

module	the name of the module
command	the name of the command
text	parameters passed to the command

Description

The command set of Sun WorkShop Visual Replay is intended for replaying user actions and for checking the state of an application with respect to its widget hierarchy and its resource settings. There is nothing to stop you adding your own commands to meet your own needs. For example:

- To produce screen dumps at various points in a replay session.
- To do other sorts of consistency checking on the widget hierarchy - one example would be to interface with Doug Young's *widgetlint* library.
- To insert a probe or a patch for a particular debugging problem. This will be of most use in a stripped optimized binary, where you do not have access to the full power of the debugger.

import allows you to load a module of your own commands into a script. Once the module has been loaded the commands in it can be invoked using the *user* command. You can import as many modules as you wish.

Example

```
import mymodule
in ApplicationShell
     cascade file_menu
           select fm_print
           in print_dialog
                   user myscreendumper print_dialog
```

Notes

The shell and setenv interface is the preferred route if the actions you need to perform do not involve extensive access to the widget hierarchy, or inspection of the internals of your program. In the latter case, see “Adding Your Own Sun WorkShop Visual Replay Commands” on page 471 to see how to add your own commands to Sun WorkShop Visual Replay.

Sun WorkShop Visual Replay Widget Naming Conventions

In Sun WorkShop Visual Replay, the widget name is what you use to reference a widget. One of the main tasks for any widget-based testing tool is identifying the right widget. The naming convention must be unambiguous, without being over-complicated.

Here are the rules used by Sun WorkShop Visual Replay:

1. If the control is a widget (i.e. not a gadget), and it is the only widget with that name in the current dialog, use the widget name, e.g.

   ```
   in ApplicationShell
           push mywidget
   ```

2. If the control is a gadget, use parentname.gadgetname

   ```
   in ApplicationShell
   ```

```
push myradiobox.mytogglebuttongadget
```

3. Where a widget name is null (i.e. “”), use unnamed, e.g.

```
in ApplicationShell
        push myradiobox.unnamed
```

4. Where there are multiple instances of this widget name (or gadget name) in the current shell, then reference the instance by number, e.g.

```
in ApplicationShell
        push mywidget#17
        push myradiobox.unnamed#3
        push myradiobox#2.unnamed#2
```

5. If you are writing your script by hand, the `tree` command can be used to examine the widget hierarchy:

```
in ApplicationShell
        tree ApplicationShell
```

This outputs a recursive listing of the widget hierarchy. The listing contains the actual widget name, and in parenthesis, the name you should use for Sun WorkShop Visual Replay, if it is different from the actual name.

6. If the shell name is ambiguous, then use instances, e.g.

```
in myshell#2
      push button1
in myshell#3
      push button2
```

Note – the instance numbers are automatically calculated when you record a script. *`Instance #3`* simply means the third occurrence of that name in a depth-first left to right search of the widget hierarchy for that shell.

APPENDIX **B**

Motif XP Reference

Introduction

The Motif XP library allows you to share code between Motif and Microsoft Windows by providing a mapping of most of the MFC classes and methods to Motif widgets and X or UNIX calls. This chapter documents the library.

Using the Motif XP

To make full use of the following information, start by checking which MFC class you are dealing with. “Mapping Motif Widgets to Microsoft Windows” on page 782 will give you this information. You can then look up the class in the following pages and find which methods are available.

Note – All variables and methods beginning *xd_* are specific to the Motif XP - you can use them on Motif but not on Microsoft Windows.

For information on the MFC and the methods, you should consult your MFC documentation supplied with the environment you are using on Microsoft Windows.

Enhancing the Motif XP

The source code for the Motif XP is provided with Sun WorkShop Visual, allowing you to add to it if you wish.

It is located in $VISUROOT`/src/motifxp/lib` (where $VISUROOT is the path to the root of the Sun WorkShop Visual installation directory). Each class has a separate source file and is commented to help you find your way around. The public headers are in xdclass.h which can be found in the $VISUROOT`/src/motifxp/h` directory.

Motif XP Library

class CObject

The class *CObject* is the principal base class for the XP library. All other classes are derived from this one.

virtual ~CObject();

Destroys a CObject object.

protected CObject();

Constructs a CObject object.

virtual Widget xd_rootwidget(); and *virtual void xd_rootwidget(Widget xd_rootwidget);*

The first version of *xd_rootwidget()* returns the widget pointer of the widget at the root of the hierarchy which is represented by the *CObject* object. The second version sets the root widget.

class CFrameWnd : public CWnd

The class *CFrameWnd* provides the functionality of a Microsoft Windows single document interface overlapped or pop-up frame window. It is used by Sun WorkShop Visual to support the ApplicationShell widget.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of *XmNtitle* for the Shell widget. Returns 0 if the widget has not yet been created, otherwise returns the length of the text.

protected virtual int xd_get_window_text_length() const;

Returns the length of the widget's *XmNtitle* resource. Returns 0 if the widget has not yet been created.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the *XmNtitle* and *XmNiconName* for the Shell widget to *lpszString*.

protected virtual BOOL xd_show_window(int nCmdShow);

Used to implement `ShowWindow` for ApplicationShell. Supports `SW_SHOWMINIMIZED`, `SW_HIDE` and `SW_RESTORE` only.

class CCmdTarget : public CObject

The class *CCmdTarget* is the base class for the XP library message-map architecture. A message map routes commands or messages to the member functions you write to handle them. This Motif version includes no functionality; the class is included only for compatibility with the Microsoft Windows code.

class CWnd : public CCmdTarget

The class *CWnd* provides the base functionality of all window classes in the XP library. The following MFC methods have been implemented:

CWnd();

virtual ~CWnd();

int GetWindowText(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the window text for the widget. This is implemented by calling the virtual member function *xd_get_window_text().*

int GetWindowTextLength() const;

Gets the length of the window text for the widget. This is implemented by calling the virtual member function *xd_get_window_text_length().*

BOOL EnableWindow(BOOL bEnable=TRUE);

Enables or disables a window. Returns 0 if the widget has not yet been created, 0 if the widget was previously enabled or non-zero if the widget was previously disabled.

void SetWindowText(LPCSTR lpszString);

Sets the window text for the widget. This is implemented by calling the virtual member function *xd_set_window_text()*.

BOOL ShowWindow(int nCmdShow);

Show, iconize (ApplicationShell or TopLevelShell only) or hide a window. Returns 0 if the widget has not yet been created; 0 if the window was previously hidden or non-zero if the window was previously visible. It is implemented by calling the virtual member function *xd_show_window()*.

*void xd_call_data (XmAnyCallbackStruct *call_data);* and *XmAnyCallbackStruct *xd_call_data () { return _xd_call_data; }*

The first version of *xd_call_data()* is used by the Sun WorkShop Visual generated code to store a callback's call_data in the class. It can be retrieved in the callback method using the second version.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Used by the sub-classes to implement *GetWindowText()*.

protected virtual int xd_get_window_text_length() const;

Used by the sub-classes to implement *GetWindowTextLength()*.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Used by the sub-classes to implement *SetWindowText()*.

protected virtual BOOL xd_show_window(int nCmdShow);

Implements default show and hide behavior for ShowWindow. For gadgets it manages and unmanages the gadget, for widgets it sets *mappedWhenManaged* appropriately.

class CDialog : public CWnd

The class *CDialog* is the base class used for displaying dialog boxes on the screen. To make a useful class, you would normally derive another class from *CDialog*.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of *XmNtitle* for the Shell widget. Returns 0 if the widget has not yet been created, otherwise returns the length of the text and the text is placed into *lpszStringBuf*.

protected virtual int xd_get_window_text_length() const;

Returns the length of the widget's *XmNtitle* resource. Returns 0 if the widget has not yet been created.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the *XmNtitle* and *XmNiconName* for the Shell widget to *lpszString*.

protected virtual BOOL xd_show_window(int nCmdShow);

Implements `ShowWindow` for TopLevelShell or DialogShell. Supports `SW_SHOWMINIMIZED` (TopLevelShell only), `SW_HIDE` and `SW_RESTORE`.

class CScrollBar : public CWnd

The class *CScrollBar* provides the functionality of a Microsoft Windows scroll-bar control.

int GetScrollPos() const;

int GetPos() const;

Returns 0 if the widget has not yet been created, otherwise returns *XmNvalue*. You can use either of these routines.

void GetScrollRange(LPINT lpMinPos, LPINT lpMaxPos) const;

If the widget has been created sets *lpMinPos* and *lpMaxPos* to *XmNminimum* and *XmNmaximum* respectively.

int SetScrollPos(int nPos, BOOL bRedraw = TRUE);

int SetPos(int nPos, BOOL bRedraw = TRUE) const;

If the widget has been created sets *XmNvalue* to *nPos* and returns the previous *XmNvalue*, otherwise returns 0. You can use either of these routines.

void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE);

If the widget has been created sets *XmNminimum* and *XmNmaximum* to *nMinPos* and *nMaxPos* respectively.

void ShowScrollBar(BOOL bShow = TRUE);

If the widget has been created manages or unmanages it as determined by *bShow*.

class CFileDialog : public CDialog

The *CFileDialog* class encapsulates the Microsoft Windows common file dialog box, providing an easy way to implement File Open and File Save As dialog boxes (as well as other file selection dialog boxes) in a manner consistent with Microsoft Windows standards.

```
CFileDialog (BOOL bOpenFileDialog,
      LPCSTR lpszDefExt = NULL,
      LPCSTR lpszFileName = NULL,
      DWORD dwFlags =
      OFN_HIDEREADONLY |OFN_OVERWRITEPROMPT,
      LPCSTR lpszFilter = NULL,
      CWnd* pParentWnd = NULL);
```

The constructor simply builds a *CFileDialogObject*. The *lpszFileName* and *lpszFilter* parameters are used to set the *XmNdirSpec* and *XmNpattern* resources of the file selection box in the *DoModal()* method. The *pParentWnd* resource should point to a *CFrameWnd* object.

virtual ~CFileDialog();

Destroys the *CFileDialog* object, freeing private class variables.

virtual int DoModal();

Sets the *XmNdirSpec* and *XmNpattern* resources as specified in the constructor then executes a private event loop until the OK, Cancel, or Popdown callback is processed.

CString GetPathName() const;

Returns the value of the file selection box’s *XmNdirSpec* resource.

protected virtual void OnCancel();

Called when the user presses the Cancel button or pops down the dialog from the window menu. Sub-classes should call this method when overriding *OnCancel()* if they want the file selection to complete.

protected virtual void OnOK();

Called when the user presses the OK button. Sub-classes should call this method when overriding *OnOk()* if they want the file selection to complete.

virtual BOOL OnInitDialog();

Returns True by default. This is overridden in Sun WorkShop Visual generated code to call the create method which will create the widgets.

class CSplitterWnd : public CWnd

Used to implement PanedWindows.

class CMenu : public CObject

The class *CMenu* is a class for handling the Microsoft Windows menu control.

CMenu();

Creates a CMenu object.

~CMenu();

Destroys a CMenu object.

UINT CheckMenuItem(UINT nIDCheckItem, UINT nCheck);

Sets the check state for a menu item which corresponds to a toggle button. The *nCheck* parameter specifies both the required state of the item (MF_CHECKED or MF_UNCHECKED) and the interpretation of *nIDCheckItem* (MF_BYCOMMAND and MF_BYPOSITION). These two values should be specified using bitwise OR (e.g. *menu->CheckMenuItem (ID_toggle_b,* MF_BYCOMMAND | MF_CHECKED *)*). The function returns -1 if the menu item is not found or is not a toggle button (note that the MFC will allow any menu item to be checked - even a separator). The previous state (MF_CHECK or MF_UNCHECKED) is returned otherwise. If *nCheck* includes MF_BYCOMMAND any submenus are also searched.

UINT EnableMenuItem(UINT nIDEnableItem, UINT nEnable);

Enables or Disables a menu item. The *nEnable* parameter specifies both the required state of the item (MF_ENABLED or MF_GRAYED) and the interpretation of *nIDEnableItem* (MF_BYCOMMAND and MF_BYPOSITION). These two values should be specified using bitwise OR (e.g. *menu->EnableMenuItem (ID_toggle_b, MF_BYCOMMAND | MF_GRAYED)*). The function returns -1 if the menu item is not found or is a menubar, menu, separator or a cascade button and MF_BYCOMMAND is specified. The previous state (MF_ENABLED or MF_GRAYED) is returned otherwise. If *nEnable* includes MF_BYCOMMAND any submenus are also searched. Note that the state MF_DISABLED (insensitive but not grayed out) is not supported.

UINT GetMenuState(UINT nID, UINT nFlags)const;

Gets the state of a menu item. The *nFlags* parameter specifies the interpretation of *nID* (MF_BYCOMMAND or MF_BYPOSITION). The function returns -1 if the menu item is not found or is a separator and MF_BYCOMMAND is specified. A bitwise OR of the states (MF_CHECKED, MF_UNCHECKED, MF_SEPARATOR, MF_ENABLED or MF_GRAYED) is returned otherwise. If *nFlags* is MF_BYCOMMAND any submenus are also searched. Note that in MFC, GetMenuState for a popup menu also returns the number of items in the high order byte. This is not supported by the Motif XP.

*BOOL TrackPopupMenu (UINT nFlags, int x, int y, CWnd *pWnd, LPCRECT lpRect = 0);*

This function simulates the behavior of the MFC TrackPopupMenu function. The function retrieves the call_data from the window specified by *pWnd* (this will have been saved by the callback function). If the event in the call_data is a ButtonPress event the popup menu is positioned using the call_data's event (not the function parameters), the menu is managed and TRUE is returned. FALSE is returned otherwise.

*void xd_register_menu(CMenu *menu);*

Used by the toolkit to map IDs to menu items.

void xd_register_menu_item(UINT nIDItem, Widget item);

Used by the toolkit to map IDs to menu items.

protected Widgetxd_get_menu_item_by_position(UINT nPos);

Used by the toolkit to map IDs to menu items.

protected Widgetxd_get_menu_item_by_id(UINT nIDItem);

Used by the toolkit to map IDs to menu items.

class CComboBox : public CWnd

The class *CComboBox* is used to wrap an OptionMenu to provide an interface equivalent to the ComboBox.

int GetCurSel() const;

Returns the (zero based) index of the currently selected item. Returns 0 if the widget has not yet been created.

int GetLBText(int nIndex, LPSTR lpszText) const;

Gets a copy of the text of the item into *lpszText* identified by *nIndex* and returns its length. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range.

int GetLBTextLen(int nIndex) const;

Returns the length of the text of the item identified by *nIndex*. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range.

int SetCurSel(int nSelect);

Sets the current selection to be the item identified by *nSelect*. Returns 0 if the widget has not yet been created and LB_ERR if the index is out of range. Otherwise returns the index of the selected item. Note unlike MFC passing *nSelect* as -1 to clear the selection is not supported.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Returns the text of the selected item in *lpszStringBuf*. Returns 0 if the widget has not been created, LB_ERR if there is no selected item, the length of the text otherwise.

protected virtual int xd_get_window_text_length() const;

Returns -1 if the widget has not yet been created, and 0 if it has. This corresponds to MFC behavior.

protected virtual void xd_set_window_text(LPCSTR);

This is a noop for *CComboBox*.

class CStatic : public CWnd

The class *CStatic* implements a Microsoft Windows static control which is a simple text field, implemented with a Label widget.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the *XmNlabelString* resource for the widget to an XmString created with `XmStringCreateLocalized()` using *lpszString*.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of *XmNlabelString* for the widget into *lpszStringBuf*. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual int xd_get_window_text_length() const;

Returns the length of the *XmNlabelString* resource for the widget. If the widget has not yet been created 0 is returned.

class CButton : public CWnd

The *CButton* class provides the functionality of Microsoft Windows button controls and is implemented with either a PushButton or a ToggleButton.

int GetCheck() const;

Gets the check state of a button. Returns 0 if the widget has not yet been created, is not a toggle button or is a toggle button and is not set. Returns 1 if the toggle button is set. Note that the MFC can return a value 2 (indeterminate state) which is not supported by the Motif XP.

void SetCheck(int nCheck);

Sets the state of a toggle button according to *nCheck*. This is a noop for push buttons.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the *XmNlabelString* resource for the widget to an XmString created with `XmStringCreateLocalized()` using *lpszString*.

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the value of *XmNlabelString* for the widget into *lpszStringBuf*. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual int xd_get_window_text_length() const;

Returns the length of the *XmNlabelString* resource for the widget. If the widget has not yet been created 0 is returned.

class CBitmapButton : public CButton

The class *CBitmapButton* implements a button with a bitmap instead of text.

class CListBox : public CWnd

The class *CListBox* provides the functionality of a list box which can display a list of items that the user can view and select.

CListBox();

Initializes the private data.

virtual Widget xd_rootwidget(); and *virtual void xd_rootwidget(Widget xd_rootwidget);*

Overrides the methods in *CObject* so that the class can distinguish between List and ScrolledList.

virtual Widget xd_listwidget();

Returns the list widget for the object. For an ordinary List this is the same as the root widget, however, it is different for a ScrolledList.

int DeleteString(UINT nIndex);

Deletes the list item identified by *nIndex* (zero based). Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range or the number of items remaining in the list.

int GetCount() const;

Returns 0 if the widget has not yet been created, otherwise returns the number of items in the list (*XmNitemCount*).

int GetCurSel() const;

Gets the index of the currently selected item in a single select list (*XmNselectionPolicy* is *XmSINGLE_SELECT* or *XmBROWSE_SELECT*). Returns 0 if the widget has not yet been created, LB_ERR if the list is a multiple selection list or it has not selected item. Otherwise the index of the selected item is returned. Note that the MFC returns an arbitrary positive value if the list is a multiple selection list, Motif XP always returns LB_ERR.

int GetSel(int nIndex) const;

Returns the selection state of the item indicated by *nIndex*. Returns 0 if the widget has not yet been created or if the item is not selected. Returns LB_ERR if the index is out of range, or a positive value if the item is selected.

int GetSelCount() const;

Returns the number of selected items in a multiple selection list. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, or the number of selected items otherwise.

int GetSelItems(int nMaxItems, LPINT rgIndex) const;

Gets the indices of the selected items in a multiple selection list and copies them into the array *rgIndex*. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, or the number of indices copied otherwise.

int GetText(int nIndex, LPSTR lpszBuffer) const;

Gets the text of an item identified by *nIndex* into *lpszBuffer*. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the length of the text otherwise.

int GetTextLen(int nIndex) const;

Gets the length of the text of an item identified by *nIndex*. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the length of the text otherwise.

int GetTopIndex() const;

Returns the index of the item that is visible at the top of the list. Returns 0 if the widget has not yet been created.

int InsertString(int nIndex, LPCSTR lpszItem);

Inserts an item into the list at the position given by *nIndex*, If nIndex is -1 the item is appended at the end of the list. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, the position at which the item was inserted otherwise.

void ResetContent();

Removes all the items from a list.

int SelItemRange(BOOL bSelect, int nFirstItem, int nLastItem);

Selects or deselects, according to *bSelect*, a range of items in a multiple selection list. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list, a value other than LB_ERR otherwise.

int SetCurSel(int nSelect);

Select an item, identified by *nSelect*, in a single selection list and scroll it into view. If *nSelect* is -1, the selection is cleared. Returns 0 if the widget has not yet been created, LB_ERR if the list is a multiple selection list or the index is out of range, a value other than LB_ERR otherwise.

int SetSel(int nIndex, BOOL bSelect = TRUE);

Selects or deselects, according to *bSelect*, an item in a multiple selection list. If nIndex is -1 all items are selected or deselected. Returns 0 if the widget has not yet been created, LB_ERR if the list is a single selection list or the index is out of range, a value other than LB_ERR otherwise.

int SetTopIndex(int nIndex);

Scroll the list to make the item identified by *nIndex* visible. Returns 0 if the widget has not yet been created, LB_ERR if the index is out of range, a value other than LB_ERR otherwise.

class CEdit : public CWnd

The class *CEdit* provides the functionality of a Microsoft Windows edit control which is a rectangular window in which the user can enter text. Implemented with either a Text or TextField widget.

CEdit();

Initializes the private data.

virtual Widget xd_rootwidget(); and *virtual void xd_rootwidget(Widget xd_rootwidget);*

Overrides the methods in *CObject* so that the class can distinguish between Text and ScrolledText.

virtual Widget xd_textwidget();

Returns the text widget for the object. For ordinary Text this is the same as the root widget, however, it is different for ScrolledText.

void Clear();

Deletes the currently selected text (`XmTextRemove()`).

void Copy();

Copies the currently selected text to the clipboard (`XmTextCopy()`).

void Cut();

Deletes the currently selected text and copies it to the clipboard. (`XmTextCut()`).

void GetSel(int &nStartChar, int &nEndChar) const;

Gets the start and end of the selected text. Returns start and end as 0 if there is no selected text.

void LimitText(int nChars = 0);

Limit the number of characters that can be typed. If *nChars* is 0 the limit is set to maximum.

void Paste();

Insert data from the clipboard into the text widget (`XmTextPaste()`).

void ReplaceSel(LPCSTR lpszNewText);

Replaces the current selection with the text supplied in *lpszNewText*. If there is no selection the text is inserted at the insert cursor position.

BOOL SetReadOnly(BOOL bReadOnly = TRUE);

Sets the *XmNeditable* resource of the widget to be !*bReadOnly*. Returns 0 if the widget has not yet been created, otherwise returns 1.

void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);

Sets the current selection to the text specified by *nStartChar* and *nEndChar*. Will also set *XmNautoShowCursorPosition* to !*bNoScroll*.

protected virtual void xd_set_window_text(LPCSTR lpszString);

Sets the value for the widget to *lpszString* (`XmTextSetString()`).

protected virtual int xd_get_window_text(LPSTR lpszStringBuf, int nMaxCount) const;

Gets the text from the widget (*XmTextGetString()*) into *lpszStringBuf*. If the widget has not yet been created 0 is returned, otherwise the length of the string is returned.

protected virtual int xd_get_window_text_length() const;

Returns the length of the widget’s text. If the widget has not yet been created 0 is returned.

class CWinApp : public CCmdTarget

The class *CWinApp* is the base class from which you derive a Microsoft Windows application object for initializing your application and for running the application.

CWinApp(const char pszAppName = NULL);*

Constructs a *CWinApp* object.

const char m_pszAppName;*

The name of the application. This comes from the parameter passed to the *CWinApp* constructor.

int m_nCmdShow;

Defaults to `SW_RESTOR`.

*CWnd *m_pMainWnd;*

The main window (ApplicationShell) of the application.

*Display *xd_display();* and *void xd_display(Display *display);*

These two functions store and retrieve the applications Display connection.

*char **xd_argv() const;* and *void xd_argv(char **argv);*

These two functions store and retrieve the argv parameters passed into `main()`.

int xd_argc() const; and *void xd_argc(int argc);*

These two functions store and retrieve the argc parameter passed into `main()`.

*char *xd_app_class() const;* and *void xd_app_class(char *app_class);*

These two functions store and retrieve the application class name used in `XtOpenDisplay()`.

CWinApp* AfxGetApp()

Returns the one, and only, instance of a *CWinApp* object.

Linking Error with Some Compilers

Some C++ compilers will fail to link, producing the following sort of errors:

```
Undefined symbol
CWnd::xd_get_window_text_length(void) const
CFrameWnd::xd_get_window_text_length(void) const
CButton::xd_get_window_text_length(void) const
CButton::__vtbl
CMenu::xd_register_menu_item(unsigned int, _WidgetRec*)
CDialog::xd_show_window(int)
CDialog::xd_get_window_text_length(void) const
CDialog::xd_set_window_text(const char*)
CEdit::__vtbl
```

The Problem

The compiler requires there to be an implementation of the copy constructor.

The Remedy

If this occurs, do the following:

1. **Find the header file *xdclass.h* in $VISUROOT/src/motifxp/h (where $VISUROOT is the path to the root of the Sun WorkShop Visual installation directory).**

2. **Locate the following lines:**

```
private:
    // Certain C++ compilers (e.g. gcc 2.5) require there to be an
    // implementation of the copy constructor. If your application
    // fails to link try using the second version of the constructor
    CObject(const CObject& objectSrc);
     // no default copy
    //CObject(const CObject& objectSrc) { abort();}
     // no default copy
```

3. **Follow the instructions so that the first line beginning `CObject` is commented out and the second has the comment markers removed:**

```
     // CObject(const CObject& objectSrc);
     // no default copy
     CObject(const CObject& objectSrc) { abort();}
     // no default copy
```

4. **Add the following include line somewhere above the line containing the call to `abort`:**

```
          #include <stdlib.h>
```

5. **Recompile your generated code.**

The application should now link successfully.

APPENDIX **C**

Getters and Setters

Introduction

Getters and setters are provided for those widgets which can have a value or a state. This appendix lists the available getters and setters for each of those widgets. These routines are toolkit-independent. To use them, you must first define a Group containing the widget(s) and then set up a Smart Code callback. For more information on these subjects, see:

1. Chapter 15, “Groups”, starting on page 477.

2. Chapter 16, “Get/Set Smart Code”, starting on page 485.

3. Chapter 17, “Thin Client Smart Code”, starting on page 501.

4. Chapter 18, “Internet Smart Code”, starting on page 533.

The widgets covered here are:

- Label and Button
- Toggle
- Text Field, Text, and Scrolled Text
- Scale
- List and Scrolled List
- Option Menu
- Radio Box

How to Use This Information

A table is provided for each widget which shows the X resource which can be accessed using a getter and a setter. One is the default - this is the one which is accessed from the server. The server simply does a get or a set of the “value” of a widget. The “value” is this default.

For each widget, an example of how to use the getters and setters is given in separate subsections for C, C++ and Java code.

More Information

There are extensive online reference documents. Open the following file in an HTML browser for a list of contents:

```
$XDROOT/lib/locale/<YourLocale>/sc/index.html
```

where XDROOT is the install directory of your X-Designer and YourLocale is the locale you are using. If you are unsure about your locale, try typing locale into a terminal window. This prints out your locale information. Use the string assigned to LANG. Example locales are:

C (for English).

ja (for Japanese).

In addition, once you have generated code, you have a file called `index.html` in the directory where your code was generated. This contains hypertext links giving you access to the online reference material.

Label and Button

TABLE 28-1 Available Getters/Setters for Label and Button

Resource Name	Type	
Value	char *	Default
Sensitive	int	

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, “Get/Set Smart Code”, starting on page 485 for more details.

This example gets and sets the value of a label. label1 is a member of group mygroup.

```
char * val = SC_GET(Value,mygroup->label1);
SC_SET(Value,mygroup->label1,”my label”);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the value of a label. “label1” is a member of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
char * val = g->label1->getValue();
g->label1->setValue(“my label”);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member's class.

This example is the same as the C code example above; it gets and sets the value of a label. "label1" is a member of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
String val = g.label1.getValue();
g.label1.setValue("my label");
```

Toggle

TABLE 28-2 Available Getters/Setters for Toggle

Resource Name	Type	
State	int[1]	Default
Sensitive	int	

[1] For Java, State is boolean.

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, "Get/Set Smart Code", starting on page 485 for more details.

This example gets and sets the state (whether or not it is set) of a toggle. "toggle1" and "toggle2" are members of group mygroup.

```
int state = SC_GET(State,mygroup->toggle1);
SC_SET(State,mygroup->toggle2, state);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member's class.

This example is the same as the C code example above, it gets the state of one toggle and sets the state of another. "toggle1" and "toggle2" are members of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
int state = g->toggle1->getState();
g->toggle2->setState(state);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member's class.

This example is the same as the C code example above; it gets and sets the state of a toggle. "toggle1" and "toggle2" are members of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
boolean state = g.toggle1.getState();
g.toggle2.setState(state);
```

Note that "state" is a boolean here - for C and C++ it would be an int.

Text Field, Text, and Scrolled Text

Getters and Setters for text controls are available through the SC_GET() and SC_SET() macros, defined in groups_c/sc_types.h.

TABLE 28-3 Available Getters/Setters for Text

Resource Name	**Type**	
Value	char *	Default
Sensitive	int	

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, “Get/Set Smart Code”, starting on page 485 for more details.

This example gets and sets the value (that is, the contents) of a text. text1 is a member of group mygroup.

```
char * contents = SC_GET(Value,mygroup->text1);
SC_SET(Value,mygroup->text1,”a new string”);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the contents of a text. “text1” is a member of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
char * contents = g->text1->getValue();
g->text1->setValue(“a new string”);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above; it gets and sets the contents of a text. “text1” is a member of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
String contents = g.text1.getValue();
g.text1.setValue(“a new string”);
```

Scale

TABLE 28-4 Available Getters / Setters for Scale

Resource Name	Type	
Value	int	Default
Sensitive	int	

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting / setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, “Get / Set Smart Code”, starting on page 485 for more details.

This example gets and sets the value (that is, the number on the scale) of a scale widget. scale1 is a member of group mygroup.

```
int val = SC_GET(Value,mygroup->scale1);
SC_SET(Value,mygroup->scale1, 1);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the scale number of a scale widget. “scale1” is a member of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
int scaleValue = g->scale1->getValue();
g->scale1->setValue(1);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above; it gets and sets the scale number of a scale widget. “scale1” is a member of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
int scaleValue = g.scale1.getValue();
g.scale1.setValue(1);
```

List and Scrolled List

TABLE 28-5 Available Getters/Setters for List

Resource Name	Type	
Items	char **	
Sensitive	int	
SelectedItems	char **	Default

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, “Get/Set Smart Code”, starting on page 485 for more details.

This example gets and sets the list of selected items of the list widget. list1 is a member of group mygroup. The list of selected items is a null terminated array of strings.

```
char ** my_stringlist = SC_GET(SelectedItems,mygroup->list1);
SC_SET(SelectedItems,mygroup->list1,a_new_stringlist);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the selected items in a list widget. “list1” is a member of Group mygroup. The list of selected items is a null terminated array of strings.

```
mygroup_c * g = (mygroup_c *) getGroup();
char ** my_stringlist = g->list1->getSelectedItems();
g->list1->setSelectedItems(a_new_stringlist);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member's class.

This example is the same as the C code example above; it gets and sets the selected items in a list widget. "list1" is a member of Group mygroup. Use the Java built-in "<array>.length" to find out how many Strings there are in the array.

```
mygroup_c g = (mygroup_c ) getGroup();
String [] my_stringlist = g.list1.getSelectedItems();
int how_many = my_stringlist.length;
g.list1.setSelectedItems(a_new_stringlist);
```

Option Menu

TABLE 28-6 Available Getters/Setters for Option Menu

Resource Name	Type	
Label	char *	
Sensitive	int	
SelectionByName[1]	char *	
SelectionByIndex	int	Default

[1] This is the string being displayed and not the widget name.

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, "Get/Set Smart Code", starting on page 485 for more details.

This example gets and sets the selected item in the optionmenu. optionMenu1 is a member of group mygroup.

```
char * val = SC_GET(SelectionByName, mygroup->optionMenu1);
SC_SET(SelectionByName,mygroup->optionMenu1, ”Option 1”);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the selected item in the option menu. “optionMenu1” is a member of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
char * val = g->optionMenu1->getSelectionByName();
g->optionMenu1->setSelectionByName(“Option 2”);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above; it gets and sets the selected item in the option menu. “optionMenu1” is a member of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
String val = g.optionMenu1.getSelectionByName();
g.optionMenu1.setSelectionByName(“Option 3”);
```

Radio Box

TABLE 28-7 Available Getters/Setters for Radio Box

Resource Name	Type	
Label	char *	
Sensitive	int	
SelectionByName[1]	char *	
SelectionByIndex	int	Default

[1] This is the string being displayed and not the widget name.

C Code Example

In C code, use the SC_GET and SC_SET macros. These macros take the resource name (i.e. *what* you are getting/setting) as the first parameter and the Group component as the second. For SC_SET, the third parameter is the new value. See Chapter 16, “Get/Set Smart Code”, starting on page 485 for more details.

This example gets and sets the label of the selected item in a radiobox widget. radiobox1 is a member of group mygroup.

```
char * str = SC_GET(SelectionByName, mygroup->radiobox1);
SC_SET(SelectionByName,mygroup->radiobox1, ”The text to show”);
```

C++ Code Example

In C++, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member’s class.

This example is the same as the C code example above, it gets and sets the label of the selected item in a radiobox widget. “radiobox1” is a member of Group mygroup:

```
mygroup_c * g = (mygroup_c *) getGroup();
char * str = g->radiobox1->getSelectionByName();
g->radiobox1->setSelectionByName(“The text to show”);
```

Java Code Example

In Java, the Group is a class, the group member is a variable of this class and is a class itself and the getters and setters are methods of the Group member's class.

This example is the same as the C code example above; it gets and sets the scale number of a scale widget. "scale1" is a member of Group mygroup:

```
mygroup_c g = (mygroup_c ) getGroup();
String str = g.radiobox1.getSelectionByName();
g.radiobox1.setSelectionByName("The text to show");
```

APPENDIX **D**

Application Defaults

Introduction

Sun WorkShop Visual has its own set of application resource settings. This appendix briefly describes the resources you are most likely to change to suit your personal preference. They can be altered in or appended to any application resource file, according to the configuration of your system. For example, you might only want to change the Sun WorkShop Visual application resource file, which is:

`$VISUROOT/lib/locale/<YourLocale>/app-defaults/visu`

or

`$VISUROOT/lib/locale/<YourLocale>/app-defaults/CDE/visu`

for the CDE window manager.

$VISUROOT is the path to the root of the Sun WorkShop Visual installation directory. The locale directory, named "`YourLocale`" above, defaults to "C". If you are using a different locale, however, the directory used will have the same name as the locale you are using. See "Locale" on page 616 for more details on locales. See Appendix E, "Further Reading", starting on page 885, for some suggestions on books on the X Window System.

You may also add any of the Sun WorkShop Visual resource settings to the *.Xdefaults* file in your home directory. If you do not yet have one of these, you can create one. It can contain resource information, in the format described in this chapter, for any X Windows application. See your X Window System documentation for more details.

In this section, the resource names appear in **bold type** and the default values in *italic*. To turn these into a resource file setting, simply add a line to the appropriate resource file. For example, in the following line:

```
visu.autoSave: true
```

autoSave is the name of the resource and *true* is the setting.

The visu resource file contains many resources that are not mentioned here. Most of these only need to be changed if you are working in a language other than English or have some other special requirements. For information on resources not documented in this appendix, see the comments in the visu resource file.

A resource can have different settings for the large and small-screen versions of Sun WorkShop Visual. Use the application class name visu for the large-screen version and `small_visu` for the small-screen version. Resources set under the name visu also apply to `small_visu`, unless there is a specific setting under the name `small_visu`.

Refer to Chapter 25, “Configuration”, starting on page 701, for information on customizing the Sun WorkShop Visual interface using some resources not covered in this chapter.

General

nameFont

The font in which widget names are displayed. Default for `visu`*:*
--helvetica-medium-r-normal--12-*-*-*-*-*-*-**
Default for `small_visu`:
--helvetica-medium-r-normal--10-*-*-*-*-*-*-**

labelFontList

The font in which labels are displayed in dialogs, menus etc. Default for `visu`*:*
--helvetica-medium-r-normal--12-*-*-*-*-*-*-**
Default for `small_visu`:
--helvetica-medium-r-normal--10-*-*-*-*-*-*-**

buttonFontList

The font in which button labels are displayed in dialogs, menus etc. Default for `visu`*:*
--helvetica-medium-r-normal--12-*-*-*-*-*-**
Default for `small_visu`:
--helvetica-medium-r-normal--10-*-*-*-*-*-**

textFontList

The font in which text is displayed in text boxes, list etc. Default for `visu`*:*
--helvetica-medium-r-normal--12-*-*-*-*-*-**
Default for `small_visu`:
--helvetica-medium-r-normal--10-*-*-*-*-*-**

warnOnClose

Causes Sun WorkShop Visual to warn you if you try to close a dialog with outstanding changes. Default: *true*.

warnOnSelect

Causes Sun WorkShop Visual to warn you if you try to select a different widget with outstanding changes on the currently selected widget. Default: *true*.

dialogsTopLevel

Causes Sun WorkShop Visual to create Top Level Shells rather than Dialog Shells for the dynamic display. This alters the way that the dialogs are iconised and their stacking properties. It does not, however, affect the final application. Default: *false*.

visu**dialogs.transient:false* produces a similar effect.

smallScreen

Makes Sun WorkShop Visual use the small icons. Default: *false* for `visu`, *true* for `small_visu`.

intrinsicsHeadersPrefix

The prefix of the X intrinsics header file names. An important resource that depends on where X is installed on your system. Default: *X11*.

definitionsFileName

The file containing the specifications for the definitions which you have configured onto the palette. The value of this resource is expanded by */bin/sh* and so can contain environment variables, etc. Default: *$HOME/.xddefinitionsrc*.

Callback and Prelude Editing

callbackEditing

This resource controls whether callback editing is enabled; if it is set to false then the buttons relating to this feature do not appear. Default: *true*.

editor

This resource specifies the location of the executable program which is used for editing files. Default:

```
$BINARYROOT/lib/scripts/xd_edit
```

sunEditService

This is a boolean resource which determines whether Sun WorkShop Visual uses the Sun Edit Server or not. If not, use the "**editor**" resource described above. Default: *True*

sunEditServerPath

This specifies the path of the eserve binary, including the binary name. It defaults to NULL if not defined in the resource file, which has the same effect as setting sunEditService to false.

Microsoft Windows

windows

If this resource is set, Sun WorkShop Visual will run in Microsoft Windows Mode. See the *Cross Platform Development* chapter for further details. Default *false*.

mfcTextWarningBackground

The color used to indicate that a resource is not used in Microsoft Windows flavor code.

mfcCarriageReturn

If this resource is set Sun WorkShop Visual will generate carriage return characters as well as newline characters in files generated for Microsoft Windows. Default *true*.

mfcFourEnhancements

When this resource is set to true, Sun WorkShop Visual generates code which uses the extended create functions in MFC version 4 to give your application the 3D look and feel. This resource also tells Sun WorkShop Visual to map bitmaps onto labels and icons onto buttons, giving you 3D buttons and images that do not need to be insensitive buttons. Images handled in this way are able to find their natural size. This results in an exact representation of your Motif design. Default *true*.

Filters

objectFileSuffix

Object file suffix used in generated Makefiles. Default: *.o*.

executableFileSuffix

Executable file suffix used in generated Makefiles. Default: empty string.

uidFileSuffix

uid file suffix used in generated Makefiles. Default: *.uid*.

Generate Dialog

The Generate dialog is primed with default names for the files to be generated. The defaults used can be changed, as detailed below.

cCodeSuffix

The default suffix for the code file when generating C. Default ".c"

cppCodeSuffix

The default suffix for the code file when generating C++. Default ".cpp"

cStubsCodeSuffix

The default suffix for the stubs file when generating C. Default ".c"

cppStubsCodeSuffix

The default suffix for the stubs file when generating C++. Default ".cpp"

cExternsCodeSuffix

The default suffix for the externs file when generating C. Default ".h"

cppExternsCodeSuffix

The default suffix for the externs file when generating C++. Default ".h"

uilExternsCodeSuffix

The default suffix for the externs file when generating UIL. Default ".h"

uilCodeSuffix

The default suffix for the code file when generating UIL. Default ".uil"

cUilCodeSuffix

The default suffix for the code file when generating C for UIL. Default ".c"

cPixmapsCodeSuffix

The default suffix for the pixmaps file when generating C. Default ".h"

cppPixmapsCodeSuffix

The default suffix for the pixmaps file when generating C++. Default ".h"

uilPixmapsCodeSuffix

The default suffix for the pixmaps file when generating UIL. Default ".h"

xResourcesCodeSuffix

The default suffix for the X resources file. Default ".res"

windowsResourcesCodeSuffix

The default suffix for the Microsoft Windows resource file. Default ".rc"

backupCodeSuffix

The default suffix for the backup file used for incremental stubs file generation. Default ".bak"

defaultCodeFileName

The default base name for all files to be generated, except the makefile. If the design has been saved, the save filename is used instead. Default "untitled".

makefileCodeFileName

The default name of the makefile. Default "Makefile".

Note – You can construct your own resource name for filenames based on a language tag and the word "CodeFileName". For example: "cCodeFileName" means the code file name for C code generation.

---

cUilCodeExtension

The extension added to the code filename when generating C for UIL. Default "_uil".

cStubsCodeExtension

The extension added to the stubs filename when generating C. Default "_stubs".

cppStubsCodeExtension

The extension added to the stubs filename when generating C++. Default "_stubs".

cPixmapsCodeExtension

The extension added to the pixmaps filename when generating C. Default "_pixmaps".

cppPixmapsCodeExtension

The extension added to the pixmaps filename when generating C++. Default "_pixmaps".

uilPixmapsCodeExtension

The extension added to the pixmaps filename when generating UIL. Default "_pixmaps".

Generation

makefileTemplate

Defines the default Makefile. For details, see the *Configuration* chapter.

motifMakeTemplateFile

Points to the default makefile template for Motif flavor. Default: *$*VISUROOT*/ make_templates/motif*.

mmfcMakeTemplateFile

Points to the default makefile template for Motif XP flavor. Default: *$*VISUROOT*/ make_templates/mfc*.

javaWidgetsLib and *javaWidgetsInclude*

Placed in the link/compile lines in generated Makefiles to point to the `libjavawidgets.a` library and associated include files. Default values are *$*VISUROOT*/src/java_widgets/lib/libjavawidgets.a* and *-I$*VISUROOT*/src/java_widgets/h* respectively.

c++BaseClassHeader

Header file for C++ that contains the base class definitions. Include quotes (“”) or angle brackets (<>) as required; defaults to angle brackets. To disable, set it to an empty string. Default: *xdclass.h*.

xpmHeader

Header file for definitions required by XPM library. Include quotes (“”) or angle brackets (<>) as required; defaults to angle brackets. Default: <*xpm.h*>.

generateXFuncCLinkage

Causes Sun WorkShop Visual to generate *_XFUNCPROTOBEGIN* and *_XFUNCPROTOEND* macros around the help link externs. Default: *true*.

generateRedefineDefaultWidgetName

In C++ base class declarations, Sun WorkShop Visual generates the default value for the widget name parameter to the *create()* member function: *widget_name = NULL*. In derived classes this should not be necessary as derived classes inherit such default parameter values from their base classes. However, many compilers require it if the *create()* function is called without a *widget_name* parameter. Sun WorkShop Visual will never generate such code, but you may have existing source where this is the case. Default: *true*.

oldUIL

Use this resource to tell Sun WorkShop Visual whether or not to generate old style UIL code. If set to “true”, the old style UIL compiler code is generated, for example:

```
XmPushButton gadget ...
```

If set to “false”, the newer style code is generated. For example:

```
XmPushButtonGadget ...
```

Default: *true*.

Comments in Generated Code

javaBeginGuard and *javaEndGuard*

Text put in as part of the guard comments in generated Java code. This text is only used for readability purposes. Default:

```
Sun WorkShop Visual-generated code - do not edit >>>
```

and

```
<<< Sun WorkShop Visual-generated code ends.
```

respectively.

Help

helpKey

The key to invoke the help callback. Default: *<Key>F1*. If your keyboard has a Help key, try *<Key>Help*.

helpDir

The search path to use when looking for help files. The search path is a colon-separated list. Sun WorkShop Visual looks in these directories for the file named `entry.html`, which is the starting point for links to other help files. If you are using text help for user-defined widgets or definitions, Sun WorkShop Visual looks in this directory for a *local* subdirectory. Default: *$*VISUROOT*/lib/locale/<YourLocale>/help (where $*VISUROOT *is the path to the root of the* Sun WorkShop Visual *installation directory).*

userHelpCatString

Separator string used to build help file names for user-defined widgets and definitions. Default: “.”

Text file help for user-defined widgets is defined by file and tag pairs. The file and tag are concatenated to produce a filename relative to *helpDir/local*. The value of *userHelpCatString* is used as a separator between document and tag when creating the string. For example, if *userHelpCatString* is "." the help file is *document.tag*. An alternative setting would be "/" to produce *document/tag*. The help file is assumed to be in HTML format.

Auto Save

autoSave

Activates the auto save facility. Default: *true* (active).

autoSaveThreshold

The number of changes made before an auto save occurs. Default: *20*.

autoSaveExt

The extension to the filename added by auto save. Default: *.sav*. For example, *fred.xd* becomes *fred.xd.sav*.

Layout Editor

The following resources control colors in the Layout Editor:

formFillColor

The background color of the Layout Editor. Default: *#dededededede*.

formStrokeColor

The color used to outline the Form in the Layout Editor. Default: *Black*.

widgetFillColor

The color used to fill the boxes representing the widgets in the Layout Editor. Default: *Blue*.

widgetStrokeColor

The color used to outline the widgets. Default: *Black*.

widgetDestinationColor

The color used to denote the last selected widget when doing align and distribute operations. This is the reference widget for the operation. Default: *#9a9ae1163232*.

widgetSelectColor

The color used to denote selected widgets when doing align and distribute operations. These are the widgets that will be moved by the operation. Default: *#ecc9c9eacdda*.

attachmentColor

The color used for the lines representing attachments in the Layout Editor. Default: *Black*.

Hierarchy Colors

The following resources are used when drawing widgets in the design hierarchy:

widgetForeground

The foreground color of bitmap-type widget icons. Default: *Black*. This refers to the color in which the pixmap or icon is drawn. For bitmap icons, it must contrast with the *widgetBackground* resource. It is unused for color pixmap icons.

widgetBackground

The background color of the widget icon. This color shows through the sections of color pixmaps that are set to color *none*. Default: *#fdfdf5f5ebeb*.

highlightForeground

The foreground color used for drawing bitmap-type icons when the widget is highlighted. For bitmap icons, it must contrast with the *highlightBackground* resource. It is unused for color pixmap icons. Default: *#fdfdf5f5ebeb*.

If you have a monochrome display, the default setting may cause some icons to show as all black. In this case set it to the same value as *widgetBackground*.

highlightBackground

The background color used when the widget is highlighted. For bitmap icons, it must contrast with the *highlightBackground* resource. Default: *Red*. This color shows through sections of color pixmaps that are set to color *none*.

If you have a monochrome display, the default setting may cause some icons to show as all black. In this case set it to the same value as *widgetForeground*.

Structure Colors

The following resources control the colors that indicate widgets that are designated as structures or C++ classes. They are effective only when the “Structure colors” toggle is set:

widgetFunctionBackground

Background color for widgets designated as function structures. Default: *#ecc9c9eacdda*.

widgetStructBackground

Background color for widgets designated as data structures. Default: *#9a9ae1163232*.

widgetClassBackground

Background color for widgets designated as C++ classes. Default: *#d2dfe785f3ce*.

widgetChildrenBackground

Background color for widgets designated as Children only. Default: *#d8c0d2d1f3f2*.

Background for Definitions and Instances

The following resources control displays of definitions and instances:

widgetInstanceBackground

The background color in the tree for instances. Default: *#ecc0ecc086db*.

widgetDefinitionBackground

The background color in the tree for definitions. Default: *.#86dbecc0ecc0*

widgetInstanceBitmap

On monochrome displays only, this is the background bitmap in the tree for hierarchies that are instances. Default: *25_foreground*.

widgetDefinitionBitmap

On monochrome displays only, this is the background bitmap in the tree for hierarchies that are definitions. Default: *25_foreground*.

Workarounds

tileOriginBug

Works around problems on some servers which have difficulty displaying the tree icons. Default: *false*.

alternateFolding

Works around problem on some servers which cannot do the stippling for folding. Draws folded icons with a cross through them. Default: *false*.

xorByInvert

In order that color map cells are not used up needlessly, Sun WorkShop Visual simply does XOR drawing using whatever happens to be in the color map. This will sometimes cause interactive drawing to be invisible. You can change the mechanism to use INVERT rather than XOR which may produce different results. Default: *false.*

freeStaticColors

Some servers do not allow the application to free colors cells in a static color map (more typically they simply ignore the request). Set this resource to false if you have a display with a default visual type of static (typically a VGA type screen), and your Sun WorkShop Visual is crashing with an X error when it tries to free a color. Default: *true*.

closeColourMatching

If your server is taking an unacceptably long time to load the main Sun WorkShop Visual window, setting this resource to false will prevent lengthy searches for close color matches when colors cannot be found. This would mean, however, that the pixmaps in Sun WorkShop Visual may look strange. Default: *true*.

FrameMaker

These resources are used to specify parameters used when using FrameMaker to develop help for your application.

frameMakerBinary

The binary that is executed to display FrameMaker help files. Default: *viewer*.

frameMakerTimeout

Number of seconds to wait for FrameMaker to start up before Sun WorkShop Visual attempts to talk to it. Default: *20*.

Configuration

See Chapter 25, “Configuration”, starting on page 701 for details on configuring the palette and toolbar.

stopList

A comma separated list of widgets that are not to be on the palette. The complete list for the Motif widgets is:

```
XmDialogShell, XmMainWindow, XmMenuBar, XmPulldownMenu, XmRadioBox,
XmRowColumn, XmFrame, XmDrawingArea, XmBulletinBoard, XmForm,
XmPanedWindow, XmScrolledWindow, XmMessageBox, XmMessageTemplate,
XmCommand, XmSelectionPrompt, XmSelectionBox, XmFileSelectionBox,
XmLabel, XmPushButton, XmToggleButton, XmDrawnButton, XmArrowButton,
XmCascadeButton, XmOptionMenu, XmSeparator, XmScale, XmScrollBar,
XmTextField, XmText, XmScrolledText, XmList, XmScrolledList
```

Use class names for user-defined widgets, e.g:

```
boxWidgetClass, formWidgetClass
```

Default: empty.

pm_icons, pm_labels, pm_both

These are the three toggle buttons in the Palette Layout menu. Set one of them for the default layout. Default: visu**pm_icons.set:true*

APPENDIX **E**

Further Reading

Introduction

This section supplies further details on the books which are referred to in this manual and others which we recommend for additional reading.

We list ISBN numbers but suggest you consult your book supplier for the latest editions.

Books Mentioned in This Manual

Kernighan, B.W. and Ritchie, D.M., *The C Programming Language.* Prentice Hall, 1978

First edition 1978 ISBN 0-13-110163-3

Second edition 1988 ISBN 0-13-110362-8

Open Software Foundation, *OSF/Motif* (5 vols). Prentice Hall, 1990, 1991, 1992.

OSF/Motif Style Guide 1993 ISBN 0-13-643123-2

OSF/Motif Programmer's Guide 1993 ISBN 0-13-643107-0

OSF/Motif Programmer's Reference 1993 ISBN 0-13-643115-1

OSF/Motif User's Guide 1993 ISBN 0-13-643131-3

Application Environment/Specification (AES) User Environment, Revision C 1992 ISBN 0-13-043621-6

O’Reilly and Associates**,** *The X Window System Series* (8 vols). O’Reilly and Associates, Inc., 1988, 1989, 1990, 1991, 1992, 1993

Volume 0:1992 ISBN 1-56592-008-2

Volume 1:1992 ISBN 1-56592-002-3

Volume 2:1992 ISBN 1-56592-006-6

Volume 3M:1993 ISBN 1-56592-015-5

Volume 4M:1992 ISBN 1-56592-013-9

Volume 5:1992 ISBN 1-56592-007-4

Volume 6A:1994 ISBN 1-56592-016-3

Volume 6B:1993 ISBN 1-56592-038-4

Volume 7:1993 ISBN 0-937175-87-0

Volume 8:1992 ISBN 0-937175-83-8

Books on X and Motif

The following books are useful books on the X Window System and OSF/Motif.

Beginner/Intermediate

Berlage, Thomas, *OSF/Motif: Concepts and Programming*. Addison-Wesley, 1991. ISBN 0-201-55792-4

Jones, Oliver, *Introduction to the X Window System*. Prentice Hall, 1989. ISBN 0-13-499997-5

Rost, Randi J., *X and Motif Quick Reference Guide*. Digital Press, 1993. ISBN 13-972746-9

Young, Douglas A., *X Window System: Programming and Applications with Xt, 2nd OSF/Motif Edition*. Prentice Hall, 1994. ISBN 0-13-123803-5

Young, Douglas A., *OSF/Motif Reference Manual.* Prentice Hall, 1990. ISBN 0-13-642786-3

Intermediate / Advanced

Asente, Paul J. and Swick Ralph R., *X Window System Toolkit.* Digital Press, 1990. ISBN 1-55558-051-3

Scheifler, R.W. and Gettys, J.**,** *X Window System, 3rd edition.* Digital Press, 1992. ISBN 13-971201-1

George, Alistair and Riches, Mark, *Advanced Motif Programming Techniques*, Prentice Hall, 1993. ISBN 0-13-219965-3

A more comprehensive listing of publications is posted monthly to the X newsgroup on *usenet* by Ken Lee of DEC.

Books on C++ and Object Oriented Programming

Stroustrup, Bjarne, *The C++ Programming Language, 2nd edition*. Addison-Wesley Publishing Company, 1991. ISBN 0-201-53992-6

Young, Douglas, *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall, 1992. ISBN 0-13-630252-1

Books on Microsoft Foundation Classes

Blaszczak, Mike, *The Revolutionary Guide to MFC 4 Programming with Visual C++*. Wrox Press Ltd. ISBN 1-874416-92-3

Books on Java

Flanagan, David, *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996. ISBN 1-56592-183-6

Arnold, Ken and Gosling, James, *The Java Programming Language*. Prentice Hall, 1996. ISBN 0-201-63455-4

Books on Networking and World Wide Web

Harold, Elliotte Rusty, *JAVA Network Programming*. O'Reilly, 1997. ISBN 1-56592-227-1.

World Wide Web Consortium, *World Wide Web Journal: Key Specifications of the World Wide Web*. O'Reilly, published quarterly. ISBN (of Volume 1, Issue 2) 1-56592-190-9.

Ed Tittel, Mark Gather, Sebastian Hassinger & Mike Erwin, *World Wide Web Programming With HTML & CGI*. IDG Books Worldwide, Inc., 1995. ISBN 1-56884-703-3.

Books on Internationalization

O'Donnell, Sandra Martin, *Programming for the World: a guide to internationalization*. Prentice Hall, 1994. ISBN 0-13-722190-8

Books on CDE

Lunde, Ken, *Understanding Japanese Information Processing*. O'Reilly & Associates Inc., 1993. ISBN 1-56592-043-0

Common Desktop Environment User's Guide. Addison-Wesley.
ISBN 0-201-48951-1

Books on HTML

Graham, Ian S., *HTML Sourcebook*. John Wiley & Sons Inc., 1995.
ISBN 0-471-11849-4

Glossary

accelerator A key or key combination that immediately executes a command from a menu.

accelerator text Text that appears on the buttons of a menu to remind the user of an accelerator.

action The name (such as “Arm,” “Activate,” or “Help”) of a widget’s prescribed response to an event, as defined in the widget’s translations table. Actions are mapped to functions in the widget’s action table.

action routine A function that performs an action. Many action routines are supplied with Motif. Sun WorkShop Visual users can also write their own action routines.

action table A table associated with a widget that maps actions to the action routines that perform them.

applet An applet is a small application written in Java which is embedded in a web page and runs when the page is browsed.

application class name The name given to the Application Shell of a generated application. This name is used as a title for that Shell’s window and to identify resources that belong to that application. In Sun WorkShop Visual, the application class name is assigned at code generation time.

Application Shell The type of Shell widget that is used for the primary window in the application.

attachment A constraint fixing one side of a widget to one side of a sibling widget or to one side of its parent layout widget. Attachments can be made at a fixed distance or at a percentage of the layout widget’s dimension.

button box The area reserved for buttons at the bottom of a composite widget such as a MessageBox, DialogTemplate, or FileSelectionBox.

C++ class widget A widget in the design hierarchy that is designated as a C++ class. In the generated code, Sun WorkShop Visual defines a C++ class for the widget, with named descendant widgets as members of the class.

callback A field of a widget structure that designates a list of callback functions and a user action. When the user action occurs on that widget, the functions on the callback list are executed.

callback function One of the functions on a callback list.

callback list A group of functions (callback functions) associated with a callback.

candidate list A list of character (ideograph) choices for a given typed pronunciation produced from an input method. This normally applies only to ideographic scripts.

capture Analyze the design of a running Motif application and create a ".xd" file containing an identical design. Sun WorkShop Visual Capture does this.

CGI Common Gateway Interface. The CGI standard lays down the rules for running external programs in a Web HTTP server. External programs are called gateways because they open up an outside world of information to the server.

CGI Script A program or shell script written in any language which conforms to the CGI standard for data communication.

children The widgets that are managed by, and (if visible) contained within the boundaries of, a parent widget. In Sun WorkShop Visual, children widgets appear below their parents in the design hierarchy.

Children Only widget A widget which Sun WorkShop Visual treats as a place-holder when generating code. No code is generated for the Children Only widget itself, but code is generated for any of its descendants that are designated as data structures, function structures, or C++ classes.

circular attachment In Form layout, an attachment of Widget A to Widget B, when Widget B is attached to Widget A. Attachment of Widget A to B, B to C, and C to A, or any larger loop, is also considered circular. Circular attachments are not allowed in Motif.

class hierarchy The abstract hierarchy of widget classes in Motif. Class hierarchy is distinguished from design hierarchy.

client data The single parameter that can be passed to a callback function.

code prelude Lines of code supplied in a dialog in Sun WorkShop Visual and inserted at specific points in the generated code.

color object Association of a name with a color.

complex font object Association of a name with a list of fonts grouped together by Motif into a FontList.

compound string A Motif data structure that combines a text string with font and direction information.

config utility Generic name for visu_config.

constraint resources The resources displayed on the Constraints panel. These resources can be viewed for any child of a constraint widget.

In general, any resources of a widget that control its children's sizes or positions.

constraint widget One of the two types of widgets (the Form and PanedWindow) whose children have a Constraints panel.

construction area The drawing area on the main Sun WorkShop Visual screen in which the design hierarchy is displayed.

container widget A widget whose main purpose is to contain and organize its children.

Control The term used widely for interface objects by such systems as Windows and Java. A control is a basic user interface object such as a button or text field.

converter A function or set of functions used to convert text entries on the resource panel to numeric resource values. Needed for certain types of user-defined widgets.

Core resource panel The special resource panel that lets you set the resources of the Core, Primitive, and Manager superclasses.

Core widget The broad superclass from which all widget classes are derived.

creation procedure A function generated by Sun WorkShop Visual that creates a Shell widget with its children.

data structure A widget for which Sun WorkShop Visual generates a `typedef` for a data structure, and a creation procedure that sets up that type of structure and returns a pointer to it.

derived widget class A widget class that is below another class in the class hierarchy. The derived class possesses all attributes of the classes above it, plus specialized attributes of its own.

design hierarchy The hierarchy of individual widget instances that makes up the design for an interface. Distinguished from class hierarchy.

detail The last field in an event specification for a translation, normally used to specify which key must be pressed.

Dialog Shell The type of Shell widget used for subsidiary windows. Dialog Shells cannot be iconified independently of their parent Shells.

dither Use various patterns of black and white pixels to achieve the effect of grey coloring.

DTD A DTD (Document Type Definition) is a collection of declarations (entity, element, attribute, link, map, etc.) in SGML syntax that defines the components and structures available for a class (type) of documents.

dynamic display The working version of the design that Sun WorkShop Visual creates dynamically as you build and edit the design hierarchy.

editing area The drawing area on the Layout Editor screen in which an editable sketch of the layout is displayed.

editing modes The designated behavior of mouse button 1 in the Layout Editor, as controlled by the radio buttons on the left side of the screen. The modes include “Move,” “Attach,” “Resize,” “Self,” and “Position.”

enumeration resource A resource that has a limited set of possible values. Sun WorkShop Visual displays enumeration resources on the “Settings” page of resource panels.

event An element of user input such as a key press or button press.

fold A display command in Sun WorkShop Visual that makes the folded widget’s children not appear on the screen, thus saving space.

FontList A list of fonts grouped together so that it is possible to show different fonts in the one string. FontList is a Motif term.

font object Association of a name with a font or a list of fonts.

Form attachment An attachment of one side of a widget to one side of its parent Form at a fixed horizontal or vertical offset. The offset remains the same when the Form resizes.

function structure A widget for which Sun WorkShop Visual produces a separate creation procedure in the generated code.

gadget An alternative version of certain widgets derived from the Primitive class. Unlike widgets, gadgets do not require the internal creation of a window for each instance. Use of gadgets instead of widgets may or may not be advantageous, depending on your system.

GIF Graphic Interchange Format. A file format for storing images. GIFs only store 8 bits of color information per pixel which makes them less attractive than other formats.

graying out Fuzzy display of an icon, pushbutton, or menu option. In Sun WorkShop Visual, graying out denotes that the command is inactive.

hard-wired resource A resource value that is generated into the source code, not into the X resource file. Hard-wired resource values cannot be changed without remaking the application.

HTML Acronym for “HyperText Markup Language”. HTML is a public domain format which uses only printable characters and can, therefore, be created in any text editor. It is also a standard used by many applications.

HTTP HyperText Transfer Protocol. At the beginning of every URL, you see the four letters "http". They tell the Web server how your browser intends to communicate with it. When you connect to a World Wide Web server, both systems use this protocol to transfer the document from the server to your system. For more detailed information see:

```
http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html
```

Hypertext Text, graphics and other media connected through links.

IDE Integrated Development Environment. An application comprising a set of integrated tools which help you to develop programs. IDEs normally include at least a compiler, debugger and code editor.

inherit To possess the attributes of a superclass. Derived widget classes are said to inherit from their superclasses.

input focus Indicates the window or component within a window that receives keyboard input. Sometimes called keyboard focus.

instance An individual widget data structure. Widget instances are specific examples of widget classes. To instantiate a widget means to create an instance of a widget class.

Intranet A network internal to an organization. There is usually no firewall between the computers on an intranet but there probably is one between the intranet and the internet.

Java Java is a programming language with libraries specifically geared for the Internet environment. Java is highly portable, object-oriented and interpreted. It is threaded, has automatic storage management and uses exceptions.

keysym A string used to identify a key in the detail field of a translation.

layout Geometric arrangement of widgets in an interface.

Layout Editor The interactive screen editor used for setting constraint resources for the Form, BulletinBoard, or DrawingArea widget.

layout widget Any of the widgets that can be used with the Layout Editor: a Form, BulletinBoard, RowColumn, or DrawingArea.

link A pre-defined callback provided by Sun WorkShop Visual.

link function The function code executed by a link.

loose binding A resource binding which can be made general to apply to groups of widgets or the whole application.

Manager widget A broad superclass in the Motif class hierarchy, from which most container widgets are derived. The Manager widget is derived from the Core widget.

masking toggle The unlabeled toggle next to each resource in the resource panels, used to designate which resources are generated into the X resource file and which are hard-wired into the code.

MFC Microsoft Foundation Classes. A set of base classes for building user interfaces to run on Microsoft Windows.

MIME Multimedia Internet Message Extension. MIME provides a way of extending the power of Web browsers to handle graphics, sound, multimedia and anything else except text. HTML handles only text - everything else is an extension. MIME is also used for binary email attachments. Browsers recognize MIME types in categories and file types, separated by a slash (such as image/gif). If you've registered a MIME type, the browser decodes the file and launches a helper application.

mnemonic A single character (often the initial character) of a menu or menu selection, which initiates the selection when the menu is displayed and the character is pressed on the keyboard.

module heading Lines of code inserted at the beginning of the generated primary module and the stubs file.

modifier list A field in the event specification of a translation that specifies whether modifier keys (such as <*Ctrl*> and <*Shift*>) must be pressed or not to cause the event.

module prelude 1) Lines of code inserted just after the Sun WorkShop Visual generated header and *#include* directives.

2) A general term meaning either a module prelude or module heading.

monitor When replaying and recording scripts - display the replay commands as they are recorded or replayed.

multiple selection The selection of more than one widget from the construction area.

offset The fixed distance, in pixels, between two attached widgets, or between a widget and the side of the layout widget to which it is attached.

originate Said of an attachment. The origin of an attachment is the widget from which the attachment was made.

package A term used in connection with Java code. A package is like a library in C. It provides a way to group together related object files. Each source file should be labelled with the name of the package to which it belongs. The package name is based on the directory containing the source file. Source files can "import" packages that they wish to use.

page selector The options menu at the top of some resource panels that lets you move from one page of resources to another.

parent A widget that manages and determines the layout of its children. In Sun WorkShop Visual, parent widgets are shown above their children in the design hierarchy.

pause When replaying scripts - stop replaying and wait at the current point in the script.

pixmap object Association of a name with a pixmap.

position attachment Attachment of one side of a widget at a specified percentage of the width or height of its parent Form. This type of attachment adjusts to the current Form dimensions.

pre-create prelude A code prelude inserted just before the given widget is created.

prelude A piece of code inserted into the generated files. See pre-create prelude, pre-manage prelude, module prelude.

pre-manage prelude A code prelude inserted after the given widget is created but before it is managed. Commonly used to set up client data for callbacks.

primary module The code module generated by Sun WorkShop Visual that contains the creation procedures for your interface.

primary selection In the context of the layout editor - the primary selection is the widget which was selected last. When aligning widgets, any other selected widgets will be aligned to the primary selection.

Primitive widget In the Motif class hierarchy, a broad superclass from which all the button-type widgets and several other classes are derived. The Primitive widget is derived from the Core widget.

Proxy A proxy server is a system that caches items from other servers to speed up access. On the Web, a proxy first attempts to find data locally, and if it’s not there, fetches it from the remote server where the data resides permanently.

Query String As used in this document, a query string refers to the part of a URL which appears after a question mark (?). The query string has a standard format which is understood by applications such as search engines.

radio buttons Toggle buttons grouped inside a RadioBox, or inside a Menu or RowColumn with the “Radio behavior” resource set to “Yes.” Only one radio button in the group may be selected at a time.

resource A settable field in a widget data structure. Resources control many aspects of a widget’s appearance and behavior. Resources can be set by the designer or the user, or both.

resource panel An interactive screen in Sun WorkShop Visual that lets you specify resource values for a widget.

resource prelude A prelude inserted at the beginning of the generated X resource file. Commonly used to add loose resource bindings for the entire application rather than for individual widgets.

script When running in record mode, the replay tool creates scripts containing a description of the actions performed on an application. In replay mode, the replay tool can read scripts and perform the actions described therein. Scripts are plain text and consist of keywords and variables.

secondary selection In the context of the layout editor - the secondary selection is any widget other than the primary selection. When aligning widgets, those which are the secondary selection will be aligned to the primary selection.

selected widget The widget whose icon is currently highlighted in the design hierarchy. A widget must be selected before anything can be done to it in Sun WorkShop Visual.

Server The business end of a client/server setup, a server is usually a computer that provides the information, files, Web pages, and other services to the client that logs on to it. The word server is also used to describe the software and operating system designed to run server hardware. The client/server setup is analogous to a restaurant with waiters and customers.

SGML Standard Generalized Markup Language is a data encoding that allows the information in documents to be shared -- either by other document publishing systems or by applications for electronic delivery, configuration management, database management, inventory control, etc. Defined in ISO 8879:1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

simple font object Association of name with a single font.

single step When replaying scripts - execute the next command only.

source file One of the code files generated by Sun WorkShop Visual. When contrasted to the “resource file,” this term refers to the primary module.

status line (1) An area at the bottom left of the main Sun WorkShop Visual screen and the Layout Editor which provides information relating to the menu option, toolbar button or widget at the current cursor location.

(2) When an input method is active, an extra text line appears in the active window (usually at the bottom left). This informs the user of the current mode with respect to the input method along with any other information the input method wishes to convey.

stubs file A generated file containing *#include* statements, function declarations and empty braces for callbacks.

subclass A widget class that is derived from another class.

superclass A widget class from which another class is derived.

tag Hypertext help information, consisting of a document name and a hypertext marker within the document.

template Used in AppGuru to describe an example user interface to be used as the starting point of a new design.

Thin Client A client is the “customer” side of a client/server setup. For example, to download something from an ftp site, you use ftp client software. A *thin* clint refers to a clint which only provides the thin layer of user interface. *All* of the “business” is performed in the server.

thumbnail sketch A small pictorial representation of a dialog which appears in the Sun WorkShop Visual Capture dialog when a dialog is captured.

tight binding A resource binding in which a widget has been forced to be included.

Top level Shell The type of Shell widget used for primary windows in a design other than the main application window.

translation A mapping of an event, such as a key or button press, or a sequence of events, to an action.

translations table The list of translations associated with a widget.

user action A predefined set of events, such as keystrokes or button presses, that triggers a callback.

variable name The name used to identify a widget’s data structure in the generated code. This is a C variable name, so it must not be the same as the variable name of any other widget, any other variable name or function name in your application, or any C code word.

widget One of the predefined data structures in the Motif toolkit, or other toolkits, that are used as building blocks for graphical user interfaces.

widget attachment An attachment of one widget to another within the Form.

widget class A specific type of widget.

widget name (1) The name used to distinguish a widget instance in the X resource file. This name does not have to be the same as the variable name and does not have to be unique.

(2) The name used to refer to a widget by Sun WorkShop Visual Replay. Multiple instances of the same widget name are referenced by number.

widget palette The area on the main Sun WorkShop Visual screen that shows icons representing the available widget classes.

window holding area The area at the upper right of the main Sun WorkShop Visual screen that displays one Shell icon for each window in the design.

work area The central area of a composite widget such as a MessageBox, DialogTemplate, or FileSelectionBox, which can contain one child widget of any type.

X resource file An editable file generated by Sun WorkShop Visual, containing some or all explicit resource values for the design.

XmString The Motif compound string structure.

Index