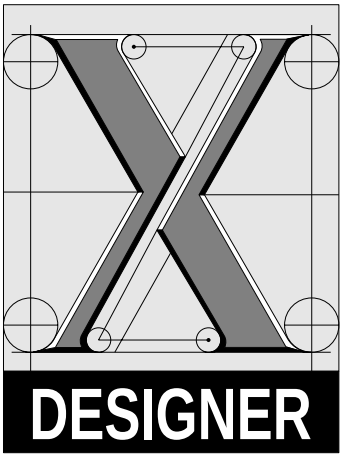


X-Designer

Release 8

User’s Guide to the New Features

Imperial Software Technology Limited
Kings Court
185 Kings Road
Reading
Berkshire RG1 4EX
Tel: +44 118 958 7055
Fax: +44 118 958 9005

email: sales@ist.co.uk
support@ist.co.uk
URL: http://www.ist.co.uk

US Office
883 Shoreline Boulevard
Suite D-220
Mountain View
CA 94043
Tel: +1 650 919 0200
Fax: +1 650 335 1054

email: sales@ist-inc.com
support@ist-inc.com
URL: http://www.ist-inc.com

X-Designer Release 8
Release 8 User's Guide to the New Features
Issue 4 November 2007

Contents

CHAPTER 1

Introduction

1.1 X-Designer 8 and Java

Java in X-Designer 8 includes support for JDK 1.4 through 1.6. The generated code has, where appropriate, removed deprecation and uses the Java Generics.

In addition, X-Designer 8 supports on the palette two new Widget classes, in aid of Java mapping:

- Spring Layout
- Box Layout[1]

The new widget classes are described in Chapter 2: The New Widgets.

1. Box Layout was introduced in JDK 1.2, but was missed in the mapping. X-Designer 8 corrects the coverage.

Where appropriate, resource panels reflect the newly supported versions of the Java toolkit. Annotations may read “1.4, 1.4+, 1.5+, 1.6” in addition to other Java annotations, reflecting whether a given feature is supported in JDK 1.4 only, JDK 1.4 through to 1.6, JDK 1.5 onwards, or JDK 1.6 only. An example of the latter can be found on the resource panels for the Spring:

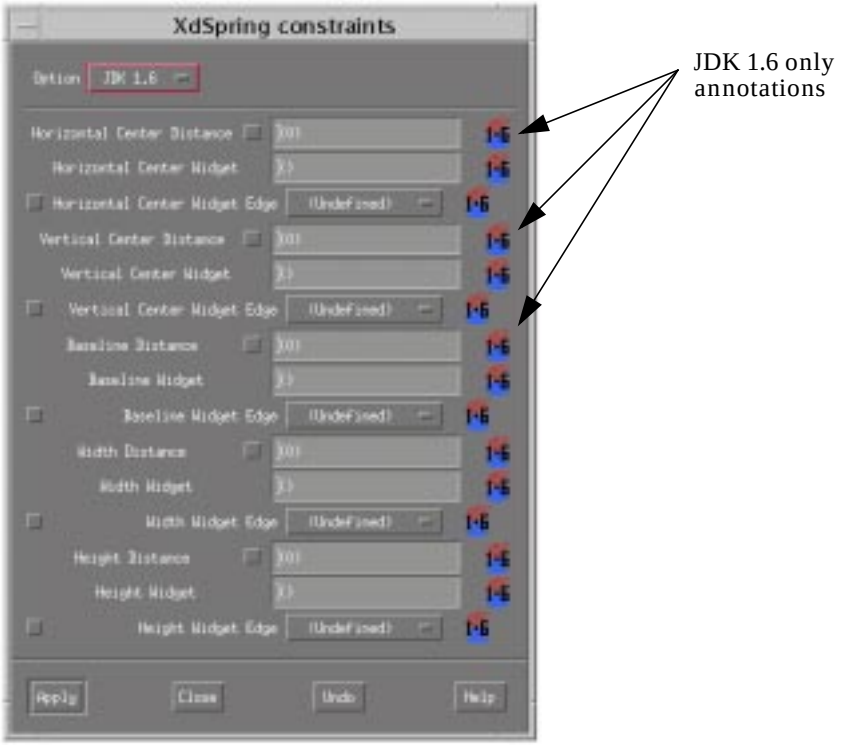


FIGURE 1-1 Java Resource Annotations for JDK 1.6

In addition to the JDK 1.4 Spring and JDK 1.2 Box Layout widgets, the following features are supported in the mapping to Java:

JDK 1.5+

- Context (popup) menus are generated and attached via the JComponent setComponentPopupMenu() method.
- The RowColumn resource XmNpopupEnabled is mapped to the JComponent setInheritsPopupMenu() method, where the value is XmPOPUP_AUTOMATIC, or XmPOPUP_AUTOMATIC_RECURSIVE.

- Component show() and hide() methods are marked as deprecated in the JDK 1.5, and accordingly these are replaced by the setVisible() method.

JDK 1.6

- The Spring Layout Horizontal Center, Vertical Center, Baseline, Width, and Height layout properties are added.

1.2 Generating Swing Versions

When generating Java Swing code, X-Designer 8 allows you to choose between versions 1.2 and 1.3 (as a single option), 1.4, 1.5, or 1.6. Do this in the Java Options Dialog which is displayed by pressing the “Java options” button in the Code Generation Dialog.

1.3 Design Time Support of Custom Hierarchies

Custom Attributes

Motif and Xt define a set of built-in resources for each of the Widgets in the toolkit. The mechanisms for setting component resources are flexible in that values may be set either in code or externally in resource files.

But the set of resources for any component is fixed by the widget author. Occasionally there is a need to be able to define application specific resources over and above the built-in set, and to piggyback the conversion mechanisms so that these application or custom attributes can also be set either in code or externally.

The most obvious case is the Motif DrawingArea, which is little more than a dumb canvas - the drawing is entirely application defined, and there are no resources to be able to specify things such as line color or textual font for the application picture.

This is where the Custom Attributes feature can assist. We can define new resources, and have X-Designer generate the code required to fetch the values of the resources from external resource files as required.

Custom Attributes are described more fully in Chapter 3: Design Time Support of Custom Hierarchies.

Override Attributes

Given that there may be multiple contexts in which the same custom attributes may apply, it may also be true that each context requires distinct values for each or any of such attributes. Following on from the discussion above, we may have for example multiple Drawing Areas, each supporting the same customised line color attribute, but each requiring different values for the color.

This is where Override Attributes provides the solution. We can create a base Definition which supports the chosen Custom Attributes, but instantiate each into our design, overriding the base default values.

Override Attributes are described more fully in Chapter 3: Design Time Support of Custom Hierarchies.

Resource Restrictions

In designing user interfaces, it is not uncommon for the work to be shared among members of a team, human factors experts designing the initial screens before handing them over to the engineers to complete the dynamics, for example. Much care may have gone into specifying a given hierarchy to conform to company or product style, or to perform specific layout and dynamic behaviour for the chosen application domain.

In this respect, there are flaws in this model in that there is nothing to prevent an engineer modifying certain aspects of the design which the original designer, who may have more experience in certain matters, considers sacrosanct.

To this end, Resource Restrictions are added to the system. It is possible for a designer of a custom hierarchy to specify that:

- certain resources may not be modified by users of the hierarchy
- certain resources may or may not have specific values applied
- the structure of the hierarchy cannot be modified by the addition of new components
- the hierarchy may not be relaid out

Resource Restrictions are described more fully in Chapter 3: Design Time Support of Custom Hierarchies.

User Documentation

In X-Designer 8 it is possible to add documentation to the generated output. We can add comments at the top of the code files, associate specific comments with specific Widgets in the Hierarchy, and document the functions and structures which we ask X-Designer to generate for us.

Documentation is stored in a language-independent manner, and will be output in appropriate ccdoc or javadoc form depending on the specific choice of generation.

User Documentation is described more fully in Chapter 3: Design Time Support of Custom Hierarchies.

1.4 Usability Enhancements

Smart Layout

The Form Layout editor is enhanced to provide standard layout styles at the push of a button - Smart Layout. Currently, the supported automatic layout types are:

- Panel Layout
- Button Box Layout

In addition, migration controls are added to the Form Layout editor, since from experience gained in cross-development environments, additional features are required in order to perform reasonable layout from imported external sources.

Smart Layout is described in Chapter 4: Usability Enhancements.

Pixmap Editor

The Pixmap editor is enhanced with new import and effect operations. In particular:

- Jpeg formatted files can be read into the editor

- The sample image can be displayed in a distinct window, or in a separate tabbed area of the editor
- Images can be rotated 90 degrees clockwise or anti-clockwise

Changes and enhancements to the Pixmap editor are described in Chapter 4: Usability Enhancements.

Outliner Widget

The Motif XmContainer is a highly capable component, providing as it does multiple types of MVC style layouts onto its children. The resource panels are particularly rich, in order to control the various layout behaviors, but for something like a simple tree layout, much has often to be set by the programmer before the tree of children (XmIconGadgets) can be constructed.

For this reason, an additional convenience object is added to the Widget Palette - the Outliner, which is an XmContainer pre-configured for tree (XmOUTLINE) layout.

Widget Transformations

Designs are not static by nature. New features are added as products progress through their normal life cycles. This will mostly take the case of new dialogs, and additional options in menus and so forth, but occasionally a new feature means that an existing functionality implemented through some specific Widget class or classes needs to be upgraded to a new paradigm.

In addition, in preparing for certain cross-platform environments, a given implementation may not map particularly well. This is particularly true when considering the Motif Geometry Managers - the Form being the most awkward case; other toolkits simply don’t have anything like the Form. To effect a map that will behave in both of the toolkits under consideration, the Form often has to be replaced when preparing the translation process.

These tasks can be achieved through cut and paste - remove the old components, add and configure the new - but this is not entirely convenient, especially as it is often the case that many of the settings of the old paradigm are still useful in the new, and these have to be re-constructed.

For this reason, Widget Transformations are introduced: selected widgets can be replaced *in situ*, keeping as much of the old settings as is possible in the mapping.

Widget Transformations are described in Chapter 4: Usability Enhancements.

Reset from Top

It is not always convenient, given a large design, to work in a particular context deep in the current hierarchy, then have to reset from the shell and revert back to the current context thereafter.

For this reason, the Reset From Top option is added to the Widget menu, which has precisely this effect: it reconstructs the entire hierarchy, and remembers and restores the current context. Reset from Top is described more fully in Chapter 4: Usability Enhancements.

1.5 X11R6 Hook Object

The Xt Hook Object was introduced into X11R6 to support runtime monitoring, debugging, and interception of Xt internal events.

X-Designer 8 allows you to use the X11R6 Hook Object directly in your applications. Although not a Widget directly added to the X-Designer palette, it can be programmed through callbacks in order to centralise the monitoring of toolkit control flow as it affects all other Widgets in the Hierarchy. Among other things, it is possible to centralise the monitoring of:

- Widget creation and destruction
- Widget change of state or set values
- Focus tracking
- Geometry Management

The Hook Object support is described in Chapter 5: The X11R6 Hook Object.

1.6 OpenGL Support

X-Designer releases prior to Version 8 support OpenGL through the standard OpenGL drawing areas[1], which can be integrated onto the Widget Palette using the XDconfig utility.

While this is sufficient for simple usage of OpenGL onto a prepared canvas, it leaves certain issues to the programmer, some of which are rather complex to get right.

- selecting and applying a Visual appropriate for OpenGL usage

1. integration kits for both the Mesa and SGI GL drawing areas are distributed with X-Designer as standard.

Prior to X-Designer 8, code to select an OpenGL-compliant Visual required preparation of a hand-crafted main module using GL specific code.

- using the transparent pixel

Further hand-crafted code was previously required to ensure that the selected GL visual also supported the transparent pixel.

X-Designer 8 rectifies these issues by providing controls that allow the programmer to request code to perform both of the above tasks automatically. If any GL drawing area is integrated onto the palette, X-Designer presents new areas in the Module menu Visuals dialog allowing the user to specify what is required.

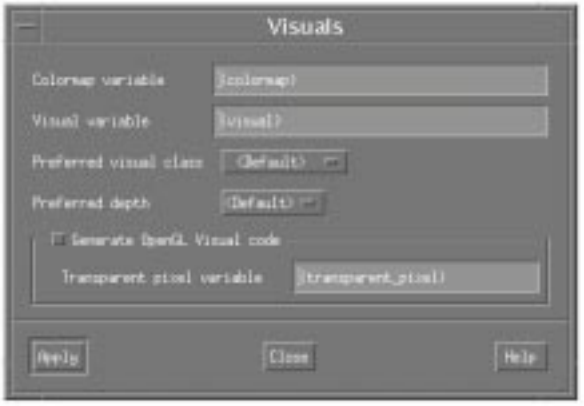


FIGURE 1-2 Visuals Dialog - OpenGL controls

Selecting the “Generate OpenGL Visuals code” toggle will cause X-Designer to generate a modified Visual fetch algorithm into the main module of your application, such that the Visual applied to all shells in the design (and therefore inherited by all components) is OpenGL compliant.

Setting the “transparent_pixel” variable field to a non-default value will also cause X-Designer to generate code that fetches a Visual with transparent pixel support, loading the value into the variable supplied. The pixel can then be used throughout the application for whatever effects are required. For example, it is possible to make a dialog “see through” by applying the transparent pixel as background to widgets in the hierarchy, thereby creating bespoke overlay effects.

1.7 Import File Formats

XML

X-Designer Release 7 can save XML for the design, but can not directly read this back. This is rectified, and “Import XML” is available from the “Import...” submenu of X-Designer’s main File menu. There is no longer the requirement to indirectly convert XML to the X-Designer save file format.

The generated XML has had an overhaul, and issues relating to non-Motif Widgets on the palette are addressed.

UIL

X-Designer Release 7 can generate UIL for the design, but can not directly read this back. This is rectified, and “Import UIL” is available from the “Import...” submenu of X-Designer’s main File menu. There is no longer the requirement to indirectly convert UIL to the X-Designer save file format.

1.8 Cross Platform Tools

xml2xd

The XML generation from X-Designer is enhanced and debugged. Although X-Designer can read back its own generated XML directly, it is recommended that XML be regenerated prior to import.

In particular, the XML for Java Emulation Widget constraints was in error; this is addressed, but fresh XML is required in order to correctly handle the constraints on import, otherwise this will affect cross-platform generation to Java.

xml2xd behaves in the same way as other X-Designer stand-alone utilities such as uil2xd. It reads from the standard input and writes to the standard output. Hence to convert generated XML to an X-Designer save file, the following command line is required:

```
xml2xd < foo.xml > foo.xd
```

CHAPTER 2

The New Widgets

2.1 Introduction

The following new Java-compatible widgets are incorporated into X-Designer:

- Box Layout
- Spring Layout

The section below, “Using The New Widgets”, illustrates how the widgets are used in X-Designer by providing a simple step-by-step tutorial. The following section, ”Widget Reference“ on page 18, is a more detailed description of each widget, together with a list of the widget’s resources.

2.2 Using The New Widgets

The following steps show you how to build a simple design using some of the new widgets.

1. **Start your design with a SessionShell.**

   SessionShell is an X11 R6 widget - it is not a Motif widget. For compatibility with earlier applications, it is a subclass of the Motif ApplicationShell. SessionShell provides support - mainly in the form of callbacks - for communication between an application and the X session manager. This allows you to control what happens when messages are received from the session manager. X-Designer 8 provides support for the SessionShell as if it were a Motif widget. We recommend you use it as your main application shell.

2. **Add a Box Layout widget to your Shell, then add three Buttons and a ComboBox to the Box.**

   The Box Layout widget is a component which lays its children out in a row column arrangement, in a single column or row, depending upon resources which configure the orientation.

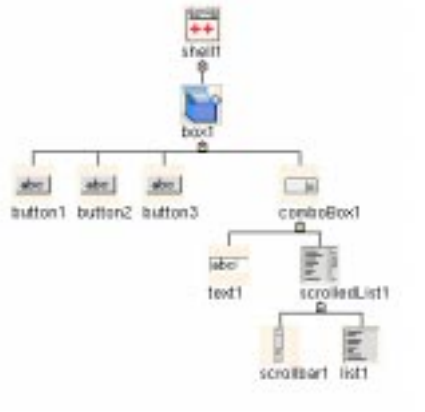


FIGURE 2-1 Box Layout: the Design Hierarchy

The Buttons and ComboBox will be horizontally positioned in the Box, centered vertically on the tallest child, in this case the ComboBox. Compare and contrast with the Motif RowColumn, where all children when laid out horizontally would be forced to be the same height.

FIGURE 2-2 Box Layout: the Dynamic Display in the default X Axis Layout

3. **Display the resources for the Box - resources are very simple, and are confined to a single “Settings” page.**

4. **Change the “Axis” resource to “Y Axis”. Press “Apply” and close the resource panel.**

   The buttons and ComboBox will now be vertically positioned in the Box, centered round the widest child. Again, this differs from the Motif RowColumn, which in vertical orientation forces all children to be the same width.

FIGURE 2-3 Box Layout: the Dynamic Display in Y Axis Layout

5. **Display the Code Generation dialog, then press the “Generate” button.**

   Three files are generated:

   1. untitled.c
   2. untitled.h
   3. Makefile

6. **Go to the directory where your files were generated and type Make into your terminal window.**

   This builds your tutorial application called “untitled”.

7. **Run the “untitled” application from your command line and check that it behaves in exactly the same way as the dynamic display in X-Designer.**

8. **Start a new design. Add a Session Shell.**

9. **Add a Spring Layout Widget to the Session Shell.**

10. **Add Four Buttons to the Spring Layout Widget.**

    The Spring Layout widget is a component which lays its children out using constraints, which specify the relative distances of child edges from each other.

    The Spring Widget constraint resources are based around the primary points of the compass: we can attach the North, South, East, and West edges of children to each other.

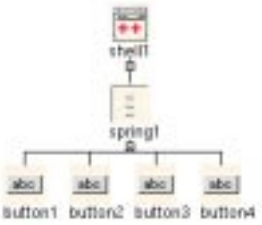


FIGURE 2-4 Spring Layout: The Design Hierarchy

    Note that in the dynamic display, all of the children will be placed on top of each other. This is not a bug, but is consistent with the Java Spring Layout class: the layout of each child has to be explicitly programmed.

11. **Explicitly set the variable names of the Spring widget and the buttons.**

    Name the Spring widget “spring1”, and the buttons “button1”, “button2”, “button3”, and “button4” respectively.

12. **Select the first button, “button1”, in the Design Window.**

13. **Display the constraint resources for the button.**

    There are three pages of resources on the panel: Dimensions, JDK 1.4+, and JDK 1.6.

14. **Display the JDK 1.4+ page in the constraints resource panel.**

    There are four groups of resources, for specifying the relative attachments and distances for each of the North, South, East, and West edges of the selected child of the Spring.

- Imagine initially we want to place button1 30 pixels from the top, and 30 pixels from the left edge of the Spring. Steps 15 to 22 below achieve this

15. **Set the North Distance to 30.**

16. **Set the North Widget to “spring1”.**

17. **Set the North Widget Edge to “North”.**

    It is the North (top) edge of the spring we are attaching from the North (top) of button1.

18. **Set the West Distance to 30.**

19. **Set the West Widget to “spring1”.**

20. **Set the West Widget Edge to “West”.**

    It is the West (left) edge of the spring we are attaching from the West (left) of button1.

21. **Apply the changes in the resource panel.**

22. **Reset the Dynamic Display from the top.**

    This is to force the Spring widget to recalculate its size based on new constraints applied to its children.

FIGURE 2-5 Spring Layout: Button1 placed 30 pixels from the North and West edges of the Spring

- Imagine now that we want to place button2 25 pixels to the right of button1, again positioned 30 pixels from the top of the spring. Steps 23 to 31 below achieve this

23. **Select button2 in the Design Window.**

24. **Set the North Distance to 30.**

25. **Set the North Widget to “spring1”.**

26. **Set the North Widget Edge to “North”.**

    It is the North (top) edge of the spring we are attaching from the North (top) of button2.

27. **Set the West Distance to 25.**

28. **Set the West Widget to “button1”.**

29. **Set the West Widget Edge to “East”.**

    It is the East (right) edge of the button1 we are attaching from the West (left) of button2.

30. **Apply the changes in the resource panel.**

31. **Reset the Dynamic Display from the top.**

    Button2 should now be to the right of button1.

FIGURE 2-6 Spring Layout: Button2 placed 25 pixels to the right of Button1

- Imagine now that we want to place buttons 3 and 4 in a second row, directly underneath buttons 1 and 2, thereby forming a kind of square grid arrangement

32. **Select button3 in the Design Window.**
33. **Set the North Distance to 30.**
34. **Set the North Widget to button1.**
35. **Set the North Widget Edge to “South”.**

    We want the North (top) of button3 to be placed relative to the South (bottom) of button1.

36. **Set the West Distance to 30.**
37. **Set the West Widget to “spring1”.**
38. **Set the West Widget Edge to “West”.**

    It is the West (left) edge of the spring we are attaching from the West (left) of button3.

39. **Apply the changes in the resource panel.**
40. **Reset the Dynamic Display from the top.**

Button3 should now be positioned underneath button1.

FIGURE 2-7 Spring Layout: Button3 placed underneath Button1

41. **Select button4 in the Design Window.**
42. **Set the North Distance to 30.**
43. **Set the North Widget to button2.**
44. **Set the North Widget Edge to “South”.**

    We want the North (top) of button4 to be placed relative to the South (bottom) of button2.
45. **Set the West Distance to 25.**
46. **Set the West Widget to “button3”.**
47. **Set the West Widget Edge to “East”.**

    It is the East (right) edge of button3 we are attaching from the West (left) of button4.
48. **Apply the changes in the resource panel.**

49. Reset the Dynamic Display from the top.

Button4 should now be placed underneath button2, and to the right of button3.

2.3 Widget Reference

The following sections provide a full widget reference for each of the new widgets, including a list of the widget's resources and a description.

Box

Resources

Settings
Axis

Description

Box is a manager widget which lays out its children in a row column arrangement. All children are displayed in a single row or column, depending on the orientation. Unlike the standard Motif RowColum, however, children are not forced to have a constant size, but are allowed to find their natural dimension centered on the widest or tallest child.

The choices for the “Axis” resource are:

- X Axis. This causes the children to be displayed in a single horizontal row, components centered on the tallest child
- Y Axis. Components are displayed in a single vertical column, centered around the widest child
- Line Axis. As for Y Axis, except the Box can also behave in a similar fashion to Flow Layout depending upon how wide or tall the Box is
- Page Axis. Also vertical in layout, except that the alignment of children will depend upon any ContainerOrientation property set in the Java containment hierarchy. Not implemented in Motif terms, since there is no equivalent of the inheritable ContainerOrientation property from any arbitrary parent of the Box, but the value is maintained here for Java consistency, and if set will behave as expected in the generated code

Spring

Resources

Dimensions

Minimum Width
Minimum Height
Maximum Width
Maximum Height
Preferred Width
Preferred Height

JDK 1.4+

North Distance
North Widget
North Widget Edge
South Distance
South Widget
South Widget Edge
East Distance
East Widget
East Widget Edge
West Distance
West Widget
West Widget Edge

JDK 1.6

Horizontal Center Distance
Horizontal Center Widget
Horizontal Center Widget Edge
Vertical Center Distance
Vertical Center Widget
Vertical Center Widget Edge
Baseline Distance
Baseline Widget
Baseline Widget Edge
Width Distance
Width Widget
Width Widget Edge
Height Distance
Height Widget
Height Widget Edge

Constraint Resources

Dimensions

Minimum Width
Minimum Height
Maximum Width
Maximum Height
Preferred Width
Preferred Height

JDK 1.4+	JDK 1.6
North Distance	Horizontal Center Distance
North Widget	Horizontal Center Widget
North Widget Edge	Horizontal Center Widget Edge
South Distance	Vertical Center Distance
South Widget	Vertical Center Widget
South Widget Edge	Vertical Center Widget Edge
East Distance	Baseline Distance
East Widget	Baseline Widget
East Widget Edge	Baseline Widget Edge
West Distance	Width Distance
West Widget	Width Widget
West Widget Edge	Width Widget Edge
	Height Distance
	Height Widget
	Height Widget Edge

Description

A Spring is a manager widget which lays out its children according to relative constraints or “springs” between each child. In some ways, it is similar to a Motif Form in that edges of a child can be positioned relative to edges of a sibling, or to the Spring itself: where the Form might have top, bottom, left, or right attachments, the Spring defines relative constraints based round primary points of the compass: the North, South, East, or West edge of a child can be placed relative to the North, South, East, or West edge of another child. The only restriction being, that attachments must be made in the same plane: North or South edges can be attached relative to the North or South of other components, and not the East or West. This is also consistent with Form edge attachments, and like the Form, the Spring performs internal checks to prevent circularity of attachments between the children.

The similarity ends where it is possible to also specify maximum, minimum, and preferred dimensions for any given child: a child can be resized as the Spring itself is resized, but only within the bounds set.

In addition, the JDK 1.6 implementation also made it possible to constrain children not just by edges, but also relative to their center, width, height, or textual baseline. Textual baseline itself is not supported in Motif; the effect will take place only in generated Java code.

Lastly, a Spring can also participate in its own geometry management. Whereas in a Form we might specify that the edge of a given child is to be placed relative to the Form itself, we cannot reverse the specification and say that a given Form edge is relative to the size and position of one of its own children. This is not true of Spring Layout: exactly the same set of constraint resources available for each

child is also available for the Spring itself, so that we might specify that the East edge of the Spring should be positioned a certain distance from the East edge of one of its children.

CHAPTER 3

Design Time Support of Custom Hierarchies

3.1 Custom Attributes

Motif and Xt define a set of built-in resources for each of the Widgets in the toolkit. The mechanisms for setting component resources are flexible in that values may be set either in code or externally in resource files.

But the set of resources for any component is fixed by the widget author. Occasionally there is a need to be able to define application specific resources over and above the built-in set, and to piggyback the conversion mechanisms so that these application or custom attributes can also be set either in code or externally.

The most obvious case is the Motif DrawingArea, which is little more than a dumb canvas - the drawing is entirely application defined, and there are no resources to be able to specify things such as line color or textual font for the application picture.

This is where the Custom Attributes feature can assist. We can define new resources, and have X-Designer generate the code required to fetch the values of the resources from external resource files as required.

1. **Start your design with a SessionShell.**

   SessionShell is an X11 R6 widget - it is not a Motif widget. For compatibility with earlier applications, it is a subclass of the Motif ApplicationShell. SessionShell provides support - mainly in the form of callbacks - for communication between an application and the X session manager. This allows you to control what

happens when messages are received from the session manager. X-Designer 8 provides support for the SessionShell as if it were a Motif widget. We recommend you use it as your main application shell.

2. **Add a Form to the shell.**

3. **Add a Motif Drawing Area to the Form.**

   The design hierarchy should be as shown below:

FIGURE 3-1 Custom Attributes - the Initial Design

4. **Attach the DrawingArea up to the containing Form on all four sides.**

   You are referred to the Form Layout section of the User Guide if you are unsure how to perform Form attachments. This step is to ensure that the DrawingArea has sensible resize behavior, since by default it will be too small to see any effects we may apply. An alternative for the purposes of this example is simply to use the Core Resources panel to assign a fixed and reasonable size to the drawing area.

5. **Ensure that the Drawing Area is selected in the Design Hierarchy.**

6. Display the Custom Attributes Dialog

The Custom Attributes dialog is displayed from the Widget menu after selecting the component where new resources are to be defined. The DrawingArea should already be selected from the operations above.

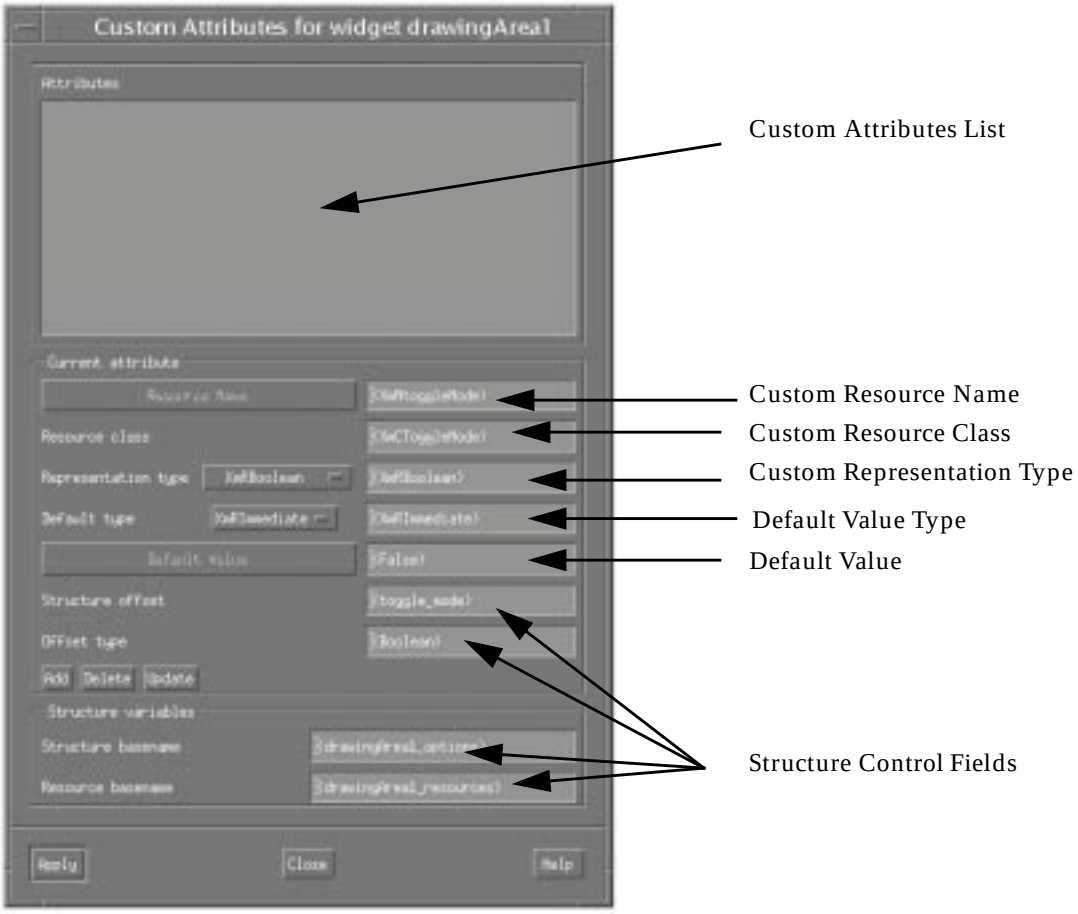


FIGURE 3-2 Custom Attributes - the Custom Attributes Dialog

By way of example, the Dialog is set up for a Boolean valued attribute, called XmNtoggleMode. This of course already exists in the Motif 2.1 sources, for the XmToggleButton and related XmToggleButtonGadget classes - it is shown here to demonstrate what a standard attribute specification might look like.

Let us suppose we want to define a line color attribute for our drawing area, and arrange for X-Designer to generate the code to fetch the value from external resources.

In order to add a custom resource, we need to define (at least) two tokens associated

with our attribute. Firstly, we need to define the resource *name*, and secondly, the resource *class*. We shall adopt the Motif naming conventions here, so that resource names are prefixed *XmN*, and resource classes with *XmC*, although there is nothing in the system which prevents you from adopting your own naming conventions.

At the top of the Custom Attributes Dialog is a list containing all of the attributes which we have defined for the currently selected widget. At present, this will be empty.

The first TextField underneath the list is the Resource Name field. This is where we will enter the name of our new attribute, although there is a popup dialog available from the left hand button, for the case where you might like to share attributes across widgets.

7. **Enter "XmNlineColor" into the Resource Name TextField.**

   By convention, the first character after the XmN prefix is lower case.

8. **Enter "XmCLineColor" into the Resource Class TextField.**

   Also by convention, the first character after the XmC prefix is upper case.

   The next step is to consider the Representation Type. This is a string which indicates the logical kind of attribute we are adding - is it a Color (Pixel), a Font, a Boolean value, and so forth.

   Motif and Xt predefine a set of built-in representation types - these are available from the option menu to the left of the Representation Type TextField.

   Clearly line color as an attribute may be represented by a Pixel in whatever code X-Designer will generate for us.

9. **Either select "XmRPixel" from the Representation Type option menu, or enter the value into the Representation Type TextField.**

   The next step is to specify a default value for the attribute. There are three ways of doing this:

   - we can specify an *immediate* value - a value that is defined straight away in the code without any levels of indirection or conversion - in which case, for the example, we need to supply a Pixel value
   - or we can supply a *string* representing the default - in which case Xt will convert the string to a Pixel for us
   - or we can supply a *procedure* - an application defined function that will return the default value

   The easiest in this case is simply to supply a string, and let Xt convert the value to a Pixel.

10. **Set the Default Type option menu to XmRString.**

11. Supply a default value into the Default Value TextField.

Any suitable color name will suffice: “red”, say.

Note that if the Default Type is XmRString, the Default Value button becomes sensitive, and X-Designer will present an appropriate dialog to select the default value - in this case, the color editor.

The final piece of the jigsaw is consideration of structure. In order to generate code which fetches external resources for widget instances, X-Designer uses the routine XtGetSubresources(). Here is not the place to discuss all the ins and outs of this routine, suffice it to say that to use this Xt function we need to define (or rather, X-Designer will define for us) two data structures:

- an XtResourceList, which specifies in Xt terms a description of how to convert the external strings found in a resource file into internal data types
- an application-specific data structure, which holds the converted data from the above

Precisely what these structures look like is again of little importance here. But the names of the structures, particularly the application specific data structure, will be important since these will be the variables which we adopt in our code to use the custom attributes.

By default, X-Designer names the structures based on the currently selected widget name. The “Structure basename” field is for naming the application-specific data structure, and the “Resource basename” field is for naming the XtResourceList variable. We can leave the resource structure name as the default, but since the application structure perhaps ought to have a fixed and known name that does not vary with the widget name, we should explicitly name the “Structure basename”, particularly if we intend to use the drawing area as a reusable Definition, which we will do so in section 3.3, “Override Attributes”.

12. Specify the “Structure basename” by entering “canvas_options” into the TextField.

What we do need to specify is Structure Offset. For each and every custom attribute we specify, X-Designer generates an element in the application specific data structure. That element will need to be declared with a variable type - the Offset Type.

The internal code representation of a color (returning to our XmNlineColor attribute) is a Pixel. X-Designer will have filled in the Offset Type field appropriately at this point.

But it won’t have named the element in the data structure. We need to choose this.

13. **Specify the Structure Offset by entering “line_color” into the Structure Offset TextField.**

    At this point, the Custom Attributes Dialog should look like this:

FIGURE 3-3 Custom Attributes - the Fields filled in

14. **Press the Add button.**

    The attribute XmNlineColor will appear in the custom attributes list at the top of the dialog.

15. **Press the Apply button, and Close the dialog.**

16. **Add an Expose callback to the drawing area, called “OnExpose”.**

    Adding callbacks is fully described in the User Manual. Callbacks are added using the “Callbacks” option from the Widget menu of X-Designer’s main menubar.

17. **Save the design as “custom.xd”.**

    Save is available from the File Menu of the X-Designer menubar.

18. **Generate code for the design.**

    You will need Code, Stubs, Externs, Main Program, and Makefile - these should all be set up in the Code Generation dialog, so simply press “Generate”.

    The first step before using the custom attribute in our code is to take a quick look at what X-Designer generates for us in the current instance.

19. **Open the file “custom.h” in your favorite text editor.**

    The file should contain, amongst other things, the following code:

```
/*
** Macro definitions for drawingArea1 Custom Attributes
*/

#ifndef   XmNlineColor
#define   XmNlineColor          "lineColor"
#endif /* XmNlineColor */

#ifndef   XmRPixel
#define   XmRPixel              "Pixel"
#endif /* XmRPixel */

#ifndef   XmCLineColor
#define   XmCLineColor          "LineColor"
#endif /* XmCLineColor */

/*
** Custom Resource Structure for canvas_options
*/
typedef struct canvas_options_s
{
        Pixel           line_color;
} canvas_options_t, *canvas_options_p;

/*
** Custom Resource Structure Variable for canvas_options
```

```
*/
extern canvas_options_p canvas_options;
```

That is, a set of conditional Macro definitions for the Custom Resource Name, Resource Class, and Representation Type, followed by the declaration of the Application-specific data structure holding the result of fetching values from external resources. Our line color is held in the *line_color* element of the *canvas_options* structure pointer.

At this point, something of the implementation must be revealed. In order to store the custom attribute structures behind the relevant widget, X-Designer uses XContext code. It cannot use XmNuserData because this is reserved for the programmer's own usage. To fetch the custom attribute structure from a widget, the convenience routine **XDget_user_context_data()** is generated by X-Designer into the application code. An external declaration of this function is also generated into the header files, as follows:

extern XPointer XDget_user_context_data(Widget w);

The return value from the function should be cast to the appropriate structure pointer type, for example:

canvas_options_p dptr = (canvas_options_p) XDget_user_context_data(w);

20. Open the file "custom_stubs.c" in your favorite text editor.

Add the following specimen code[1] to the OnExpose callback, to draw a rectangular border in the drawing area, using the *line_color* attribute:

```
void OnExpose(Widget w,
       XtPointer client_data,
       XtPointer xt_call_data)
{
       XmDrawingAreaCallbackStruct *call_data =
            (XmDrawingAreaCallbackStruct *) xt_call_data;

       static GC gc = (GC) 0; /* A graphics context for drawing */

       Dimension width = (Dimension) 0;
       Dimension height = (Dimension) 0;
       XGCValues values;
```

1. The code makes no concessions to error checking, optimization, or indeed thread safety, but is by way of example only.

```
        /* Create a Graphics Context, with Foreground set to the
        ** value of our custom attribute
        */

        if (gc == (GC) 0) {
             canvas_options_p dptr = (canvas_options_p)
                                  XDget_user_context_data(w);

        values.foreground = dptr->line_color;

                  gc = XCreateGC(XtDisplay(w),
                                 XtWindow(w),
                                 GCForeground,
                                 &values);
        }

    /* Work out the size of the drawing area at the moment */
            XtVaGetValues(w, XmNwidth, &width, XmNheight, &height,
                          NULL);

    /* Clear the drawing area */
        XClearArea(XtDisplay(w), XtWindow(w), 0, 0, 0, 0, False);

    /* Draw a rectangular border five pixels from the edge */
        XDrawRectangle(XtDisplay(w), XtWindow(w), gc, 5, 5,
                width - 10, height - 10);
    }
```

21. Type "make" from your command line prompt to build the application.

22. **Run the "custom" application.**

The application should draw a red rectangular border inside the drawing area

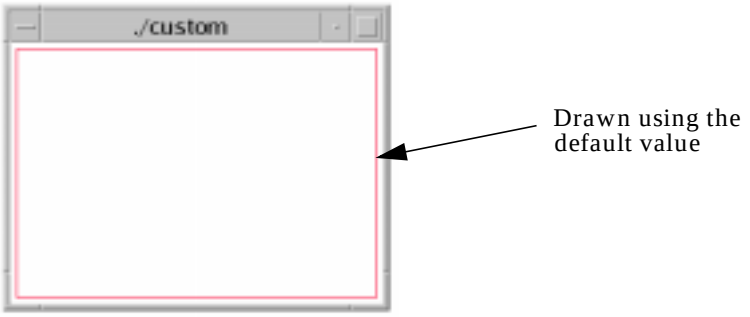


FIGURE 3-4 Custom Attributes: the Line Color Attribute in action

23. **Run the “custom” application, with external custom resources set.**

We can simulate this in a variety of ways, by editing resource files and so forth, but the simplest is to use the built-in -xrm switch on the command line. Note that any “XmN” prefix is not required when externally setting resources:

./custom -xrm “*drawingArea1.lineColor: green”

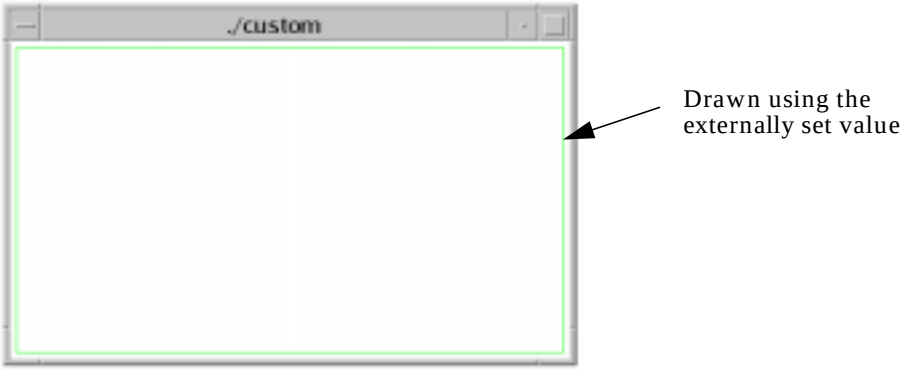


FIGURE 3-5 Custom Attributes: the Line Color Attribute externally specified

3.2 Global Custom Attributes

Just as we can define Custom Attributes on a per-widget basis, we can also define new resources globally, unassociated with any specific widget instance.

This is achieved in exactly the same way, through the Custom Attributes Dialog, except that the dialog is invoked from the Module menu.

The only difference lies in the internals of the code: X-Designer generates XtGetApplicationResources() rather than XtGetSubresources() to fetch the external values. Again, a structure, and macros, are generated into external header files, in exactly the same manner.

3.3 Override Attributes

Following on from the above example, let us suppose that we have multiple drawing areas in our design, and we would like to have a different default line color for each instance.

The simplest way to achieve this is to turn the original hierarchy containing the drawing area with the custom attribute into a *Definition* on the X-Designer Palette, and to instantiate multiple instances therefrom. This has the advantage that each instance *inherits* the custom attribute setting from the base definition - we don't have to set up custom attributes for line color for each drawing area.

In order to generate the correct code for a definition, we need to make sure we are not generating multiple spurious shells and forms, when it is only the drawing area we want on the palette.

1. **Select the Shell at the top of the design.**
2. **Display the Core Resources Panel, and select the Code Generation Page.**
3. **Using the Structure Option Menu, declare the structure to be "Children only".**

   This will prevent code from being generated for the shell itself.
4. **Apply the Core Resources Panel for the Shell.**
5. **Select the form underneath the Shell.**
6. **Using the Structure Option Menu, declare the structure to be "Children only".**

   This will prevent code from being generated for the form.
7. **Apply the Core Resources Panel for the Form.**
8. **Select the Drawing Area.**

   This is the widget we want to instantiate from the Palette.
9. **Using the Structure Option Menu, declare the structure to be "Data structure".**

   Definitions must be structured; we could equally have declared the drawing area to be a C++ class.

10. **Change the variable name of the drawing area to “baseCanvas”.**

    Definitions may not have automatically generated names, otherwise they may conflict with other designs. Any name will suffice, provided that it conforms to legal C variable name syntax. Having said that, naming the variable the same as a C library call such as “abort” or “read” would not be wise, although a modern strongly typed compiler would detect the conflict.

11. **From the Widget Menu, select the “Definition” option.**

    The Drawing Area will have a blue background in the design hierarchy.

12. **Save the design to file “baseCanvas.xd”.**

13. **Display the Code Generation Dialog.**

    At present, the dialog is set up for the previous Custom Attributes example.

14. **Press the Reset button, to rename files based around the name “baseCanvas”.**

    For a definition, we will require code, stubs, externs, but not a main module. - set the toggles to the right of code, stubs, and externs, but *not* a main module, therefore make sure the corresponding toggle is off. Also make sure that the main module name appears as default - in brackets - otherwise X-Designer will generate a Makefile rule for this file even if not generating at this time. Clear the text field if this is not so.

15. **Display the Code Options Dialog.**

    This is available from the bottom left Options button in the Code Generation dialog.

    We will not require “links” functions - in the Code Options sub-dialog presented, set the Links option menu to “None”.

    We will not require the User Attribute (XContext fetch and set) functions themselves in the base definition, although we will require external specifications. Set the “User Attributes Functions” option menu to “Declarations Only”. If we don’t do this, we can end up with multiple definitions of the X Context functions in the code files. This is no different from Link generation when creating a code base from Definitions.

16. **We are finished with the Code Options Dialog - press the “OK” Button.**

    We will require a Makefile, but it should be of “Template” kind, since we will merge the Makefile with other designs as we progress with the Override Attributes example - press the bottom right Options button, and set the “New Makefile” toggle on, and the “Template toggle” on, then apply the Makefile options dialog.

17. **Press the top right C Code Options button.**

18. **Make sure the “Include header file” toggle for baseCanvas.h is on.**

    Apply the C Code Options dialog.

19. **Press the “Generate” button to generate code for the design.**

20. **Save the design to “baseCanvas.xd” again.**

21. **From the Palette Menu, select the “Edit Definitions” option.**

    Here is not the place to discuss the ins and outs of Definitions fully - this is described in the User Guide.

22. **Press the “Prime” button.**

    This should fill in various fields based on what X-Designer can deduce from the nature of the definition. These will be sufficient for the example.

23. **Press the “Update” button.**

    We should now have an extra item at the bottom of the Widget Palette, labelled “baseCanvas”.

    We are now ready to progress to setting up Override Attributes. The “Edit Definitions” dialog can be closed.

24. **Edit baseCanvas_stubs.c in your favorite text editor.**

    Add exactly the same OnExpose callback code as in Section 3.1, step 20.

25. **Start a new design.**

    Select “New” from the X-Designer File menu

26. **Create a Session Shell.**

27. **Create a Form underneath the shell.**

28. **Add an instance of our baseCanvas to the Form, by selecting the item from the Widget Palette.**

    The design hierarchy should look likr this; note that the instantiated baseCanvas is presented in a mustard background, marking it as an instance of a user-defined palette item.

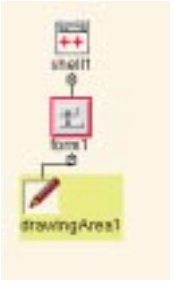


FIGURE 3-6 Override Attributes - the initial design

Initially, if you sized the base definition by attaching up to its containing Form (as opposed to assigning an explicit dimension through Core Resources), the Drawing Area will have default small size: 1 pixel by one pixel.

29. **Using the Form Layout editor, attach the drawing area on all four sides to its containing Form.**

    Or again give the drawing area a reasonable size if it has none, using the Core resources panel.

30. **Select the Drawing Area in the Widget Hierarchy.**

31. **Display the Override Attributes dialog.**

    The option is found within the X-Designer Widget menu. The following dialog should appear:

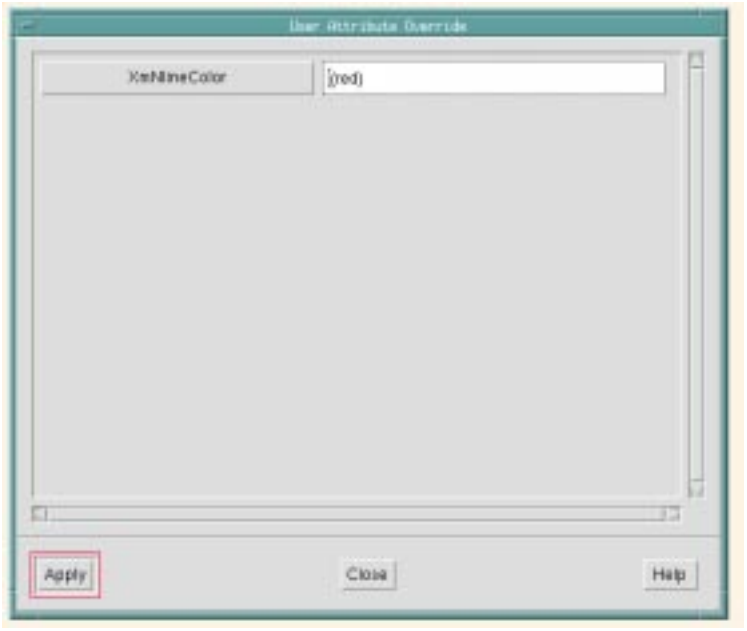


FIGURE 3-7 Override Attributes - The Override Attributes Dialog

At the top is a scrolled list containing all the attributes which are inherited from the base definition. In this case, there is only one attribute, the line color. The left hand side is a button, which if pressed will display an editor, appropriate to the custom attribute representation type, for constructing a new default value - in this case, the X-Designer color editor. The right hand side contains a text field, showing the inherited default value - “red”.

32. **Change the line color value to “yellow”.**

33. **Apply the Override Attributes Dialog.**

    Once applied, the dialog can be closed.

34. **Save the design to “overrideCanvas.xd”.**

35. **Display the Code Generation dialog.**

    This time, we do require the User Attribute (XContext) functions to be generated to code.

36. **Display the Code Options Dialog.**

    Make sure the User Attributes Functions option menu is set to “Generated to Code”[1]. Close the Code Options dialog.

37. **Display the Makefile options dialog.**

38. **Turn off the New Makefile toggle.**

    We want to merge rules for the previously generated baseCanvas into rules for the overrideCanvas code, since overrideCanvas depends on baseCanvas for the Custom Attributes feature.

39. **Turn on the Makefile Template toggle.**

    Close the Makefile options dialog once the toggles are set.

40. **Generate code, stubs, externs, Makefile, and main program.**

    Simply turn the toggles on for these types of file, and press the Generate button.

41. **Type make to build the overrideCanvas application.**

42. **Run the overrideCanvas application.**

    ./overrideCanvas

    The application should appear as follows, with the drawing area now properly yellow as the default value:

FIGURE 3-8 Override Attributes - The Running Application

Here ends the tutorial on Override Attributes, except that some tidying up is required to remove the temporary Definitions from the palette, otherwise they will continue to appear on the palette.

1. Or, Generated to Main Program. Either is acceptable.

Tidying Up

43. **From the X-Designer Palette Menu, select the “Edit Definitions” option.**
    The Edit Definitions dialog is displayed.
44. **Select “baseCanvas” in the list at the top of the Dialog.**
45. **Press the “Delete” button in the Dialog.**
    The object “baseCanvas” will be removed from the X-Designer Widget Palette.
46. **Close the Edit Definitions Dialog.**

3.4 Resource Restrictions

Given an environment in which multiple people work on the design and implementation of the application interfaces, potential for conflict arises even if modules are separated into distinct designs under the overall management by a source control system.

This is particularly acute where there are multiple designs which affect each other - where there are definitions in design A, and instances in design B, each design file being under separate file management. It does not matter what kind of control we place on the definition if the user of the instance is free to modify any or all resources in an instance.

To this end, Resource Restrictions are defined such that we may specify that users of a Definition are prevented from modifying those aspects of the original design which the designer considers sacrosanct.

Preventing Modification to the Layout of a Hierarchy

Firstly, let us consider where there are gaps in the model. We shall start by creating a simple Definition where the components are carefully laid out, and show that under normal circumstances it is possible for a user of the definition to destroy the best laid schemes.

1. **Start a new design.**
2. **Add a new Shell to the design.**

3. **Add a Form to the shell.**

   Give the Form an explicit variable name, “myRestrictedForm”.

4. **Add a Label and a TextField to the Form.**

   Give the Label the variable name “myRestrictedLabel”, and the TextField the name “myRestrictedText”.

5. **Display the Form Layout Editor.**

   Enter “50” into the Position Field.

6. **Enter “5” into the Offset Field.**

   Press the “Panel Layout” button on the Form Layout Editor toolbar.

   The Label and TextField should now be properly aligned in the Form, with built-in resize behaviour. We can assume for the purposes of this exercise that this is behaviour we do not want modified by users of this hierarchy.

7. **Declare the Form as a Data Structure.**

   The Structure option is available from the Core Resources Panel, Code Generation page.

8. **With the Form still selected in the Design Hierarchy, turn the Form into a Definition.**

   The “Definition” option is available from the Widget menu.

9. **Save the Design, to file “myRestrictions.xd”.**

   Next, we shall add the hierarchy to the Widget Palette.

10. **Display the “Edit Definitions” Dialog.**

    This is available from the Palette Menu of X-Designer’s main menubar.

11. **Press the “Prime” button.**

12. **Press the Update button.**

    A new option, labelled “myRestrictedForm” will appear at the bottom of the X-Designer palette.

13. **Close the Edit Definitions Dialog.**

    And now we instantiate our hierarchy into a new design.

14. **Start a New Design.**

15. **Add a Shell to the Design.**

16. **Add an instance of myRestrictedForm by selecting the item from the Widget Palette.**

    The Design Hierarchy should appear like this:

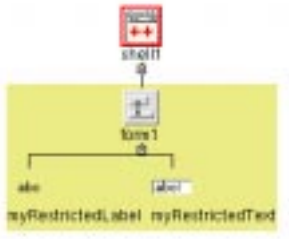


FIGURE 3-9 Resource Restrictions - the Initial Definition Instance

At this point, we can verify that the layout of our instance can be modified:

17. **Select the instance of the Form underneath the Shell.**
18. **Display the Form Layout Editor.**
19. **Move the components around, or re-attach them in any way you wish.**

    To prevent this, we need to go back to our original Definition.

20. **Open “myRestrictions.xd”.**

    X-Designer will ask whether you want to save the current design. There is no need to do so for this example.

21. **Select the Form “myRestrictedForm” in the Design Hierarchy.**
22. **Turn off the Definition state for the hierarchy.**

    The Definition toggle is in the X-Designer Widget menu.

23. Display the Resource Restrictions Dialog.

The option is available from the Widget menu of X-Designer's main menubar. The following dialog will appear:

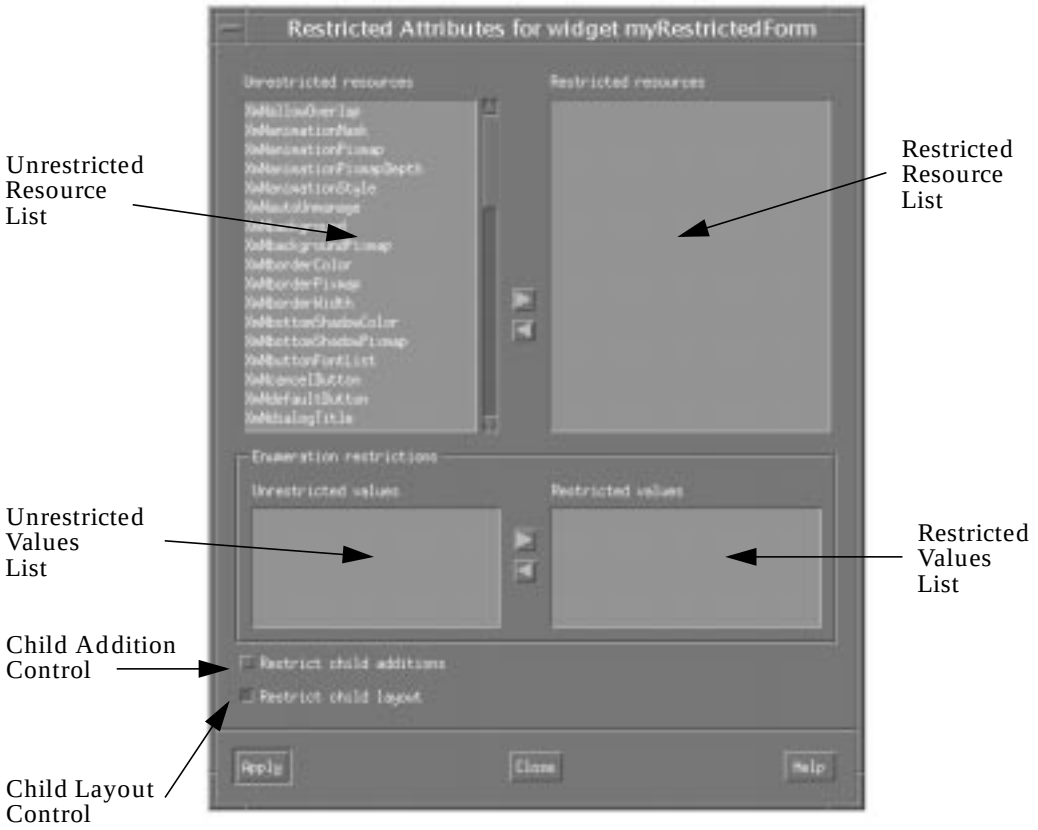


FIGURE 3-10 The Resource Restrictions Dialog - Initial State

The top of the Dialog contains two lists which allow you to specify which resources the user of a Definition may not modify.

The middle of the Dialog refines the process: specific resources can be restricted to certain values only.

We will come onto examples of using the above shortly, but for now the interest lies with the toggles at the bottom of the dialog. Using these toggles we can

- prevent a user of the Definition from adding new components to the hierarchy
- prevent a user from modifying the layout

24. Turn the "Restrict child layout" Toggle On.

25. **Turn the hierarchy back into a Definition by selecting the “Definition” option from the Widget menu.**

26. **Save the design.**

    If we now create a new design, and instantiate our hierarchy into the design, we find that it is impossible for the user to display the Form layout editor onto the hierarchy: the Layout option is now insensitive in X-Designer’s toolbar, and insensitive in the Widget menu.

    The layout of our original hierarchy can thus be preserved.

Preventing Modification to Specific Resources

If we go back to the original definition, and turn the Definition toggle back off, we can look at how it is possible to further restrict modification of the hierarchy and its properties.

27. **Open “myRestrictions.xd”.**

28. **Select the Form.**

29. **Turn the Definition toggle Off.**

30. **Select the Label “myRestrictedLabel” in the Hierarchy.**

31. **Display the Resource Restrictions Dialog for the Label.**

    X-Designer will automatically prime the fields with resources known for the Label type.

    Imagine we do not want the user of our Definition to change the XmNlabelString resource.

32. **Select XmNlabelString from the top left “Unrestricted resources” list.**

33. **Press the Right Pointing Arrow between the “Unrestricted” and “Restricted” resource lists.**

    The value XmNlabelString will disappear from the “Unrestricted” list and appear in the “Restricted” list. At this point the Dialog will appear as follows:

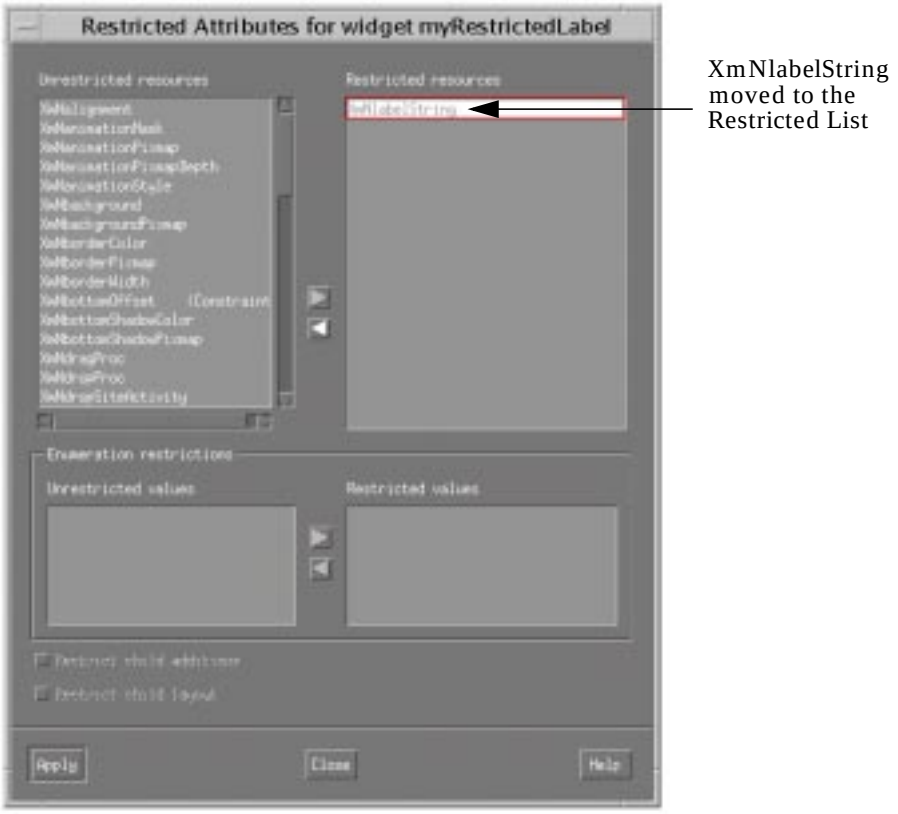


FIGURE 3-11 The Resource Restrictions Dialog: Restricting Specific Resources

34. **Apply the Dialog.**

35. **Select the Form “myRestrictedForm” in the Design Hierarchy.**

36. **Turn the Definition Toggle back on.**

37. **Save the Design.**

    If we now instantiate an instance of our “myRestrictedForm” Palette item, and select the Label within the instance we should be able to see the effect.

38. **Create a new Shell.**

39. **Add an instance of “myRestrictedForm” from the Widget Palette.**

40. **Select the Label in the Design Hierarchy.**

41. **Display the Label Resource Panel.**

    The LabelString resource will be insensitive in the Resource Panel, and therefore cannot be changed by a user of the Definition.

Preventing the Application of Specific Resource Values

For resources which are of enumerated or Boolean type, X-Designer can also prevent the selection of specific values from the enumerated set.

42. **Open “myRestrictions.xd”.**

43. **Select the Form.**

44. **Turn the Definition toggle Off.**

45. **Select the Text “myRestrictedText” in the Hierarchy.**

46. **Display the Resource Restrictions Dialog for the Text.**

    Let us imagine for the sake of argument that we don’t want the user of our hierarchy to turn off Tab Group behavior into and out of our Text widget.[1]

47. **Select XmNnavigationType in the Unrestricted List.**

48. **Press the Right Pointing Arrow between the Unrestricted and Restricted Lists.**

    XmNnavigationType will appear in the right hand list.

1. And there are good reasons why this might be the case, since it would affect keyboard navigability, and those who monitor Section 508 Accessibility for our products may find justifiable fault.

49. Now select XmNnavigationType in the Restricted List.

The set of possible values for navigation type will appear in the “Unrestricted Values” list in the middle left of the Dialog. The Dialog should appear like this:

FIGURE 3-12 The Resource Restrictions Dialog - Initial Unrestricted Values

To prevent the user of our hierarchy from turning navigation tab groups off, all we need to do is to prevent the user from setting the tab group to XmNONE.

50. Select XmNONE in the Unrestricted Values list.

51. Press the Right Pointing Arrow between the Unrestricted Values and Restricted Values list.

The value XmNONE will disappear from the Unrestricted list, and appear in the Restricted list

52. Apply the Resource Restrictions Dialog.

53. Select “myRestrictedForm”.

54. **Turn the Definition Toggle back on.**

55. **Save the Design.**

    All that remains is to check that the user of the Definition cannot set the specified value.

56. **Create a new Design.**

57. **Add a Shell to the Design.**

58. **Add an instance of “myDefinitionForm” from the Widget palette.**

59. **Select the Text Widget “myRestrictedText”.**

60. **Display the Core Resource Panel for the Text Widget, Settings Page.**

61. **Display the Navigation Type option menu.**

    The value “None” will be insensitive.

    Here ends the tutorial on Resource Restrictions, except that some tidying up is required to remove the temporary Definitions from the palette, otherwise they will continue to appear on the palette.

Tidying Up

62. **From the X-Designer Palette Menu, select the “Edit Definitions” option.**

    The Edit Definitions dialog is displayed.

63. **Select “myRestrictedForm” in the list at the top of the Dialog.**

64. **Press the “Delete” button in the Dialog.**

    The object “myRestrictedForm” will be removed from the X-Designer Widget Palette.

65. **Close the Edit Definitions Dialog.**

3.5 User Documentation

X-Designer 8 has the ability to add user-defined documentation to the design. This is maintained with the save files, and generated into the sources in a form appropriate to the chosen language. Both javadoc and ccdoc styles are implemented, although the user of the system does not have to write specific tags or comments appropriate to either: X-Designer generates the correct style automatically.

There are two aspects to automatic documented sources which X-Designer supports: data supplied by the user, and ccdoc style generated output from X-Designer itself.

Considering the latter case first, the X-Designer generated code supports automatic ccdoc and javadoc comments and directives for the functions and variables which X-Designer outputs. Clearly this can have a bearing on sources maintained under a strict code control system, where changes have to be justified. To this end, automatic documentation of X-Designer's own output, as opposed to comments directly supplied by the user, is controllable by external resource:

```
XDesigner.ccdocCommentStyle: true
```

For backwards compatibility of the X-Designer generated code, the resource is by default set to *false*, so that only directives and comments supplied by the user are operative.

There are two kinds of user-supplied documentation: module documentation, which applies to the design as a whole, and widget-specific documentation.

Module Documentation

Module documentation is generated at the top of your source files. It allows you to specify:

- Package Information
  - any package name associated with the documentation
  - any package document name
  - any package ID
- Document Information
  - the author, or authors, of the document
  - the version of the document
  - the date of the documentation
  - related document links
  - deprecated features
- Comments
  - any text you wish to illuminate the document data as a whole

Module documentation is available from the X-Designer Module menu, under the Documentation... option.

1. **Display the Module Documentation Dialog.**

   The following dialog is presented:

FIGURE 3-13 Module Documentation - Package Page

   The dialog is laid out in pages, and is also used for Widget-specific documentation, although pages and fields of the dialog are made insensitive in appropriate contexts.

   The Package and Document pages are only sensitive for Module documentation.

2. **Enter “myPackage” into the Package name field.**

3. **Enter “myDocument” into the Package document field.**

   The Package ID field is a ccdoc feature which allows you to refine the semantics of the package name. You are referred to ccdoc documentation. We shall not use it for this example.

4. **Display the “Document” page of the dialog.**

5. **Enter your name into the Author field.**

   Multiple authors can be entered, one per line: it is a multi-line TextField.

6. **Enter “1.0” into the Version field.**

7. **Enter today’s date (or indeed any date you like) into the Since field.**

   Since this is our first documentation exercise, we shall pretend there are no related documents, or deprecations. But you could enter data into these fields should you wish.

8. **Display the “Comment” page of the Dialog.**

   Enter some text into the Text box provided.

9. **Apply the dialog.**

   Display the Code Generation dialog.

10. **Generate C (or C++) Code into “myDocument.c”.**

11. **Edit the source code in your favorite text editor.**

    At the top of the file will be the data we have entered, in ccdoc style. Note that the style changes with the language generated, although the data is preserved. For example, if we generated C, the code should appear similar to the following:

```
/**
 * @pkg myPackage
 * @pkgdoc myDocument
 * @author John Smith
 * @version 1.0
 * @since September 2007
 * This comment is generated at the top of the source module,
 * and is where we can describe the contents and purpose of the sources.
 */
...
```

    And if we generated C++, the source style will be as follows:

```
//
// @pkg myPackage
// @pkgdoc myDocument
// @author John Smith
// @version 1.0
```

```
// @since September 2007
//@{
// This comment is generated at the top of the source module,
// and is where we can describe the contents and purpose of the sources.
//@}
//
...
```

This also works for other languages as well: if we generate Java, the output is appropriate for javadoc handling.[1]

Widget Documentation

For every widget in the design, we can associate some documentation data describing what it does.

12. **Add a Shell to the design.**
13. **Add a Form to the Dialog Shell.**
14. **Declare the Form to be a Data Structure.**

    Core Resources Panel, Code Generation Page.
15. **From the Widget menu of X-Designer, select “Documentation...”.**

    The Documentation dialog will appear, except that the Field page is now sensitive. For general purpose Widgets that have no structure or classing assigned, the Field page is largely redundant - documentation will be added simply through the Comments page of the dialog, and the documentation will be inserted at the point of

1. In this instance, it will be the same as the C example.

Widget instantiation in the source code. However, if we structure a component, X-Designer will generate functions with parameters, and these can be documented using the Field page. It should appear like this:

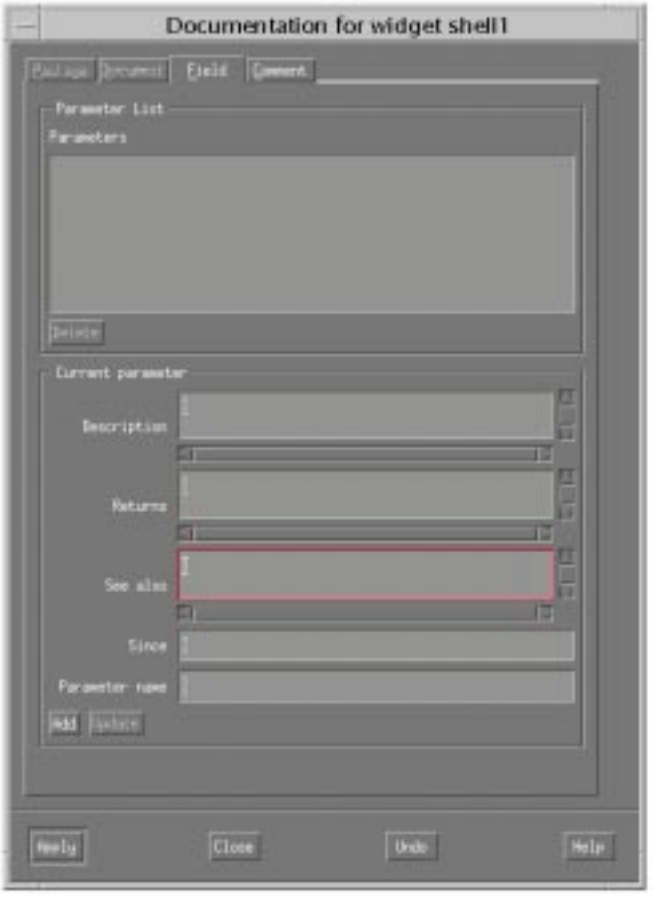


FIGURE 3-14 Widget Documentation - Field Page

For ordinary dialogs, with the exception of the primary application shell, X-Designer generates functions in the form:

```
Widget create_Hierarchy(Widget parent);
```

The primary shell creator takes four parameters:

```
Widget create_Primary(Display display,
       char *application_class,
       int  argc,
```

```
        char **argv);
```

For structures and classes, the code is as follows:

```
Structure_p create_Hierarchy(Widget parent);
```

For Definitions, the code is slightly different:

```
Widget create_Definition(Widget parent, char *name);
```

Now we could turn on automatic documentation of these procedures through the external resource:

```
XDesigner.ccdocCommentStyle: true
```

... and X-Designer will insert some parameter documentation information at the point of declaration. But the first problem here is, X-Designer doesn't know what the function or created Widget hierarchy is for, only you know this. Secondly, using code preludes, it is possible to modify the number of parameters associated with a creation procedure, adding application specific ones passed through to the code. Thirdly, X-Designer does not know in what circumstances the code is to be used if it is a reusable function, so that the parent parameter, for example, may be a specific context in the application Widget hierarchy, and perhaps should be documented as such. And lastly, X-Designer will generate comments for everything, Widget declarations, links functions, structure comments, and so on, which may simply not be required.

To this end, documentation of parameters to functions can also be done manually here.

In this instance, the form is structured, and hence there is one parameter, the parent Widget context. It returns a dynamically allocated structure containing the public components of the Form. Here, there are no sub-components, only the Form Widget itself.

16. **Enter "parent" into the Parameter name field.**
17. **Enter today's date into the "Since" field.**
18. **Enter some text into the Description field.**

    For example, "The parent Widget context. Assumed to be A, B, C."

    This describes assumptions about the parent parameter. Here there are none, but for structured code of this kind which is meant to be dynamically callable from the application, it may have some assumptions such as the contexts in which it is meant to be called, and what it is for. This is application specific.

19. **Enter some text into the Returns field.**

    For example “An instance of the Form”.

    This describes the data structure the function is returning. Again, this is application specific.

20. **Press the “Add” button.**

    The parameter list will now contain our parent parameter. The dialog should appear similar to the example picture below:

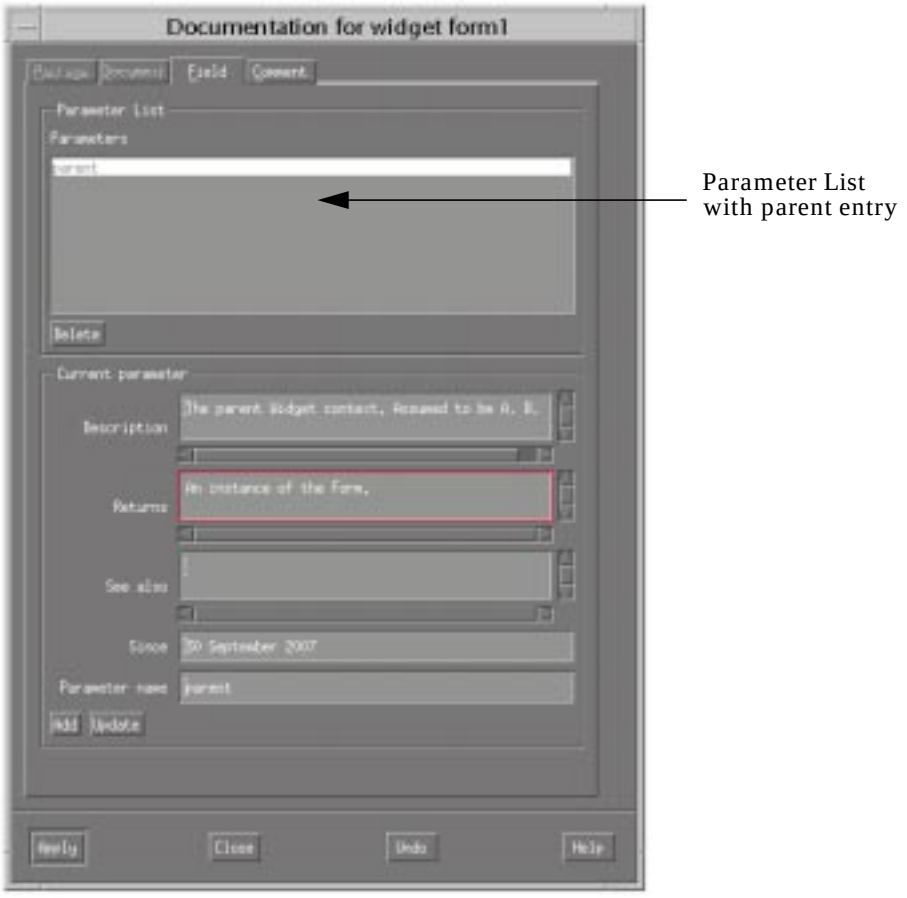


FIGURE 3-15 Widget Documentation - Example Parameter Entry

Lastly, on the Comments page, we can enter whatever text we like to fully document the Widget hierarchy at this point.

21. **Display the “Comments” page of the Widget Documentation Dialog.**

22. **Enter some text into the comments field.**

    This of course is also application specific, but typically notes would include the statement that this data structure is dynamically allocated, and it is the responsibility of the programmer to reclaim the dynamic memory through calling delete_form1()[1] at an appropriate juncture.

23. **Press the “Apply” button.**

24. **Generate a code source file in C or C++.**

25. **Open the code file, using your favorite text editor.**

    The source should have our documentation, appropriate for ccdoc in the chosen language, similar to the following:

```
/**
 * @param parent
 * The parent Widget context. Assumed to be A, B, C.
 * @returns An instance of the Form.
 * @since 30 September 2007
 * It is the responsibility of the programmer to reclaim
 * the dynamic memory through calling delete_form1() at an
 * appropriate juncture.
 */

form1_p create_form1 (Widget parent)
{
```

1. Or the appropriate generated memory reclaim routine for the given hierarchy.

CHAPTER 4

Usability Enhancements

4.1 Smart Form Layout

The Form Layout editor is enhanced to provide standard layout styles at the push of a button. Currently, the supported automatic layout types are:

- Panel Layout
- Button Box Layout

In addition, migration controls are added to the Form layout editor, since from experience gained in cross-development environments, additional features are required in order to perform reasonable layout from imported external sources. In particular, importing sources where absolute x, y, width, height directives are used to specify layout can be problematic when subsequently attempting to layout using the Form editor.

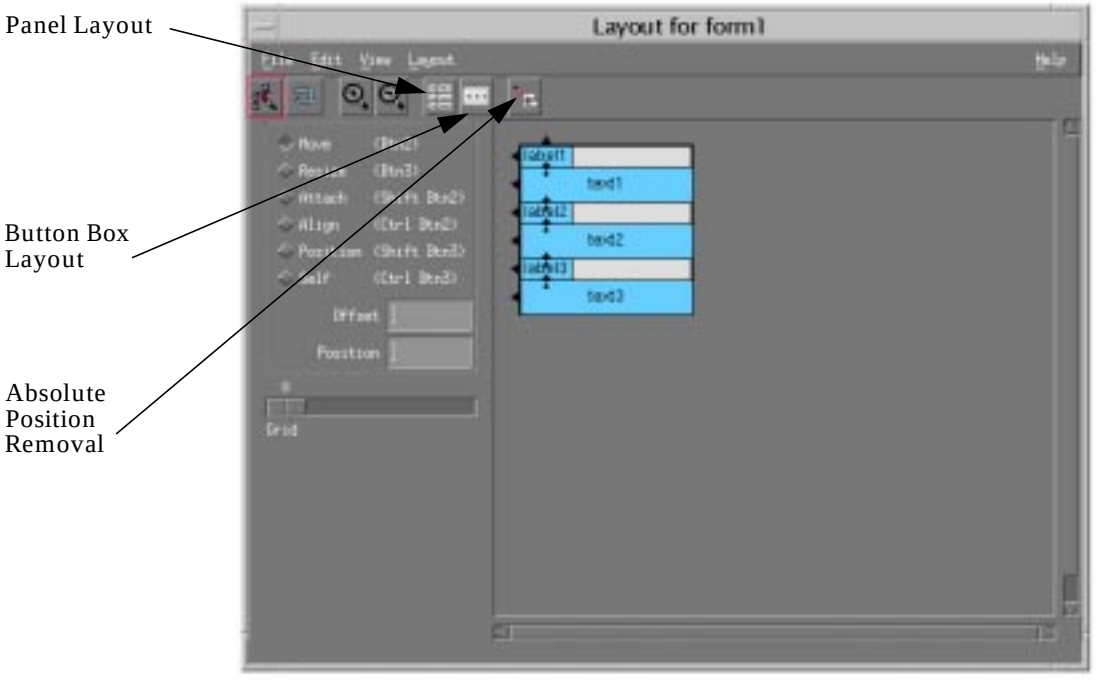


FIGURE 4-1 Form Layout Editor - Smart Layout Buttons

Panel Layout

This layout style is for creating multiple columns of aligned label/text combinations, similar to X-Designer's own resource panels. For example:

26. **Create a shell in a new design.**
27. **Add a Form to the shell.**

28. Add a number of Label/Text pairs to the Form.

Although the Form Layout editor can work with widget combinations through the selection mechanisms that are not in label/text order, for the purposes of this example it is simplest to add the children of the Form in the order Label, Text, Label, Text,...

The diagram below shows a sample widget hierarchy for this exercise:

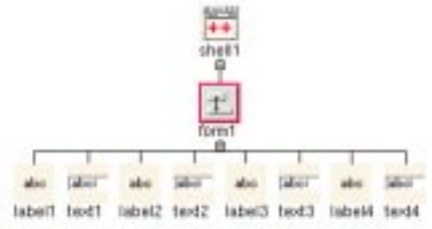


FIGURE 4-2 Form Layout Editor - Sample Design Hierarchy for Panel Layout

29. Display the Form Layout Editor.

The Layout Editor should appear similar to the figure below, depending on the number of label/text combinations you have added:

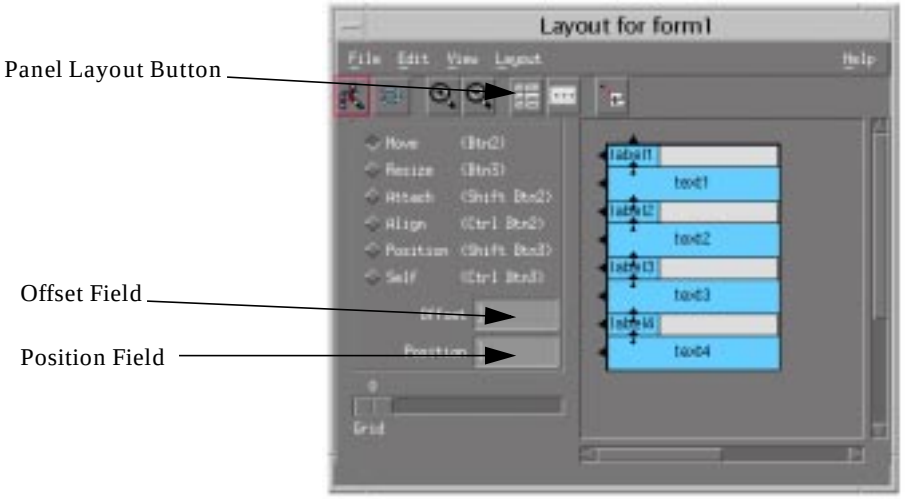


FIGURE 4-3 Form Layout Editor - Preparing the Panel Layout

30. Enter “5” into the Offset Field.

In Panel Layout, the Offset Field is used to specify gaps between the columns of items which X-Designer will create for you.

31. Enter “45” into the Position Field.

In Panel Layout, the Position Field is used to specify the location of the column separating the labels from the text fields

32. Press the Panel Layout Button.

The Form layout editor will lay out the labels and text fields into two columns, each row being 5 pixels apart vertically, and from the edge of the containing form, the column of text widgets being aligned at 45% across the form on the left edge.

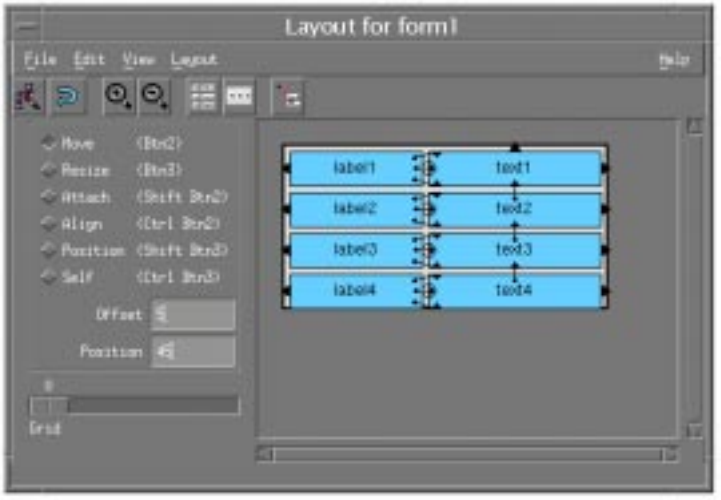


FIGURE 4-4 Form Layout Editor - Panel Layout in Action

The Dynamic Display should appear similar to the following:

FIGURE 4-5 Form Layout Editor - Panel Layout and the Dynamic Display

X-Designer adds right attachments to the TextFields, so that they have automatic resize behavior as the containing form is resized horizontally, all the time maintaining the column layout.

The Panel Layout mechanisms are fully integrated into the Form Editor “Undo” system - you can undo the entire Panel Layout at a stroke, or apply different offset / position combinations to find the proportional effect that looks best.

Panel Layout also works using Form Selection, as well as working on all of the widgets in the current form. If you select a sub-set of the children of the current form (holding down the shift key as you click on items in the editor), Panel Layout can be applied to the sub-set. This is useful where there are other control combinations that need laying out differently, but you still have individual label / text combinations where Panel Layout is preferred.

Button Box Layout

There are controls in the standard OSF Motif widget set which will give you a Button Box layout (a series of buttons laid out proportionally across the screen, with proportional resize behaviour): this can be achieved with a DialogTemplate at its simplest, but this is not always an ideal solution particularly where you want to put the button box as part of a larger arrangement.

The Form can also perform Button Box layout, if you are cunning with the use of the Fraction Base resource, and position attachments. But it is an awkward combination to set up. This is where the automatic Button Box Layout is useful: all of the calculations and positions are calculated for you, and all you need to do is add the buttons.

1. **Create a shell in a new design.**
2. **Add a Form to the shell.**
3. **Add Four PushButtons to the Form.**

   The design hierarchy should appear like this:

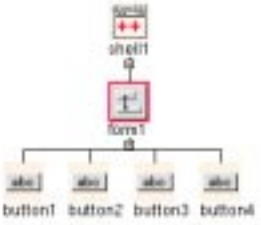


FIGURE 4-6 Form Layout Editor - Sample Hierarchy for Button Box Layout

4. **Display the Form Layout Editor.**

5. **Put “5” into the Offset Field.**

6. **Press the Button Box Layout Button.**

   The Push Buttons will be laid out horizontally across the screen, proportionally spaced, aligned 5 pixels from the top of the containing form.

FIGURE 4-7 Form Layout Editor - The Dynamic Display in Button Box Layout

Similarly to Panel Layout, Button Box Layout is integrated into the Undo system, and the Form Editor can work on sub-sets of the current Form’s children when applying Button Box Layout.

Absolute Position Removal

Components with hard coded position or dimension are antithetical to the spirit of the XmForm, where the layout algorithms work best if components are allowed to find their own natural size, and positioned using Form constraints.

Importing sources built on a BulletinBoard model - where position and size do indeed require explicit programming - can be tiresome since all of the hard coded positions and dimensions really need to be removed before layout in a Form, otherwise the settings can conflict with whatever layout one tries to set up in the Form.

To this end, the Form editor has an extra button on the toolbar, which if pressed will internally remove all position and dimensions set on children of the currently selected Form. Thereafter, Form layout can proceed normally.

4.2 Pixmap Editor

The Pixmap editor is enhanced with new import and effect operations. In particular:

- Jpeg formatted files can be read into the editor
- The sample image can be displayed in a distinct window, or in a separate tabbed area of the editor, thereby reducing the space required to display the image under construction in the editor itself

- Images can be rotated 90 degrees clockwise or anti-clockwise

When importing JPeg image files, additional controls are added to allow the user to specify dithering or best fit algorithms. Jpeg images can be particularly color rich, and can flood the colormap.

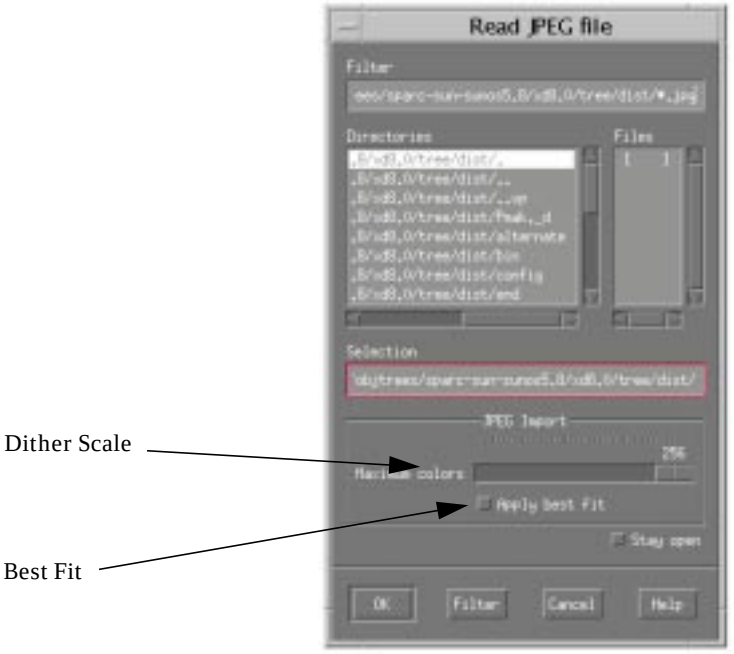


FIGURE 4-8 The Pixmap Editor - Import Jpeg

Dithering

When loading a Jpeg file, X-Designer calculates the size of the current Colormap, and offers this as an upper bound on the colors to load from the Jpeg image. If you specify, using the Maximum Colors scale, a value less than the size of the Colormap, X-Designer will load that number from the Jpeg image.

Apply Best Fit

In addition to dithering, X-Designer can apply a best-fit algorithm to assigning loaded colors from the image. That is, assuming that there are more colors in the image than the Colormap can satisfy, colors from the image will be morphed into colors in the Colormap using a best-fit closeness of RGB strategy. This may cause

different results each time the functionality is invoked, since it depends on the current Colormap state. For optimal results, you are advised to close down color-intensive tools that share the default Colormap before importing Jpeg files, or to run X-Designer with a private Colormap installed.

4.3 Outliner Widget

The Motif XmContainer is a highly capable component, providing as it does multiple types of MVC style layouts onto its children. The resource panels are particularly rich, in order to control the various layout behaviors, but for something like a simple tree layout, much has often to be set by the programmer before the tree of children (XmIconGadgets) can be constructed.

For this reason, an additional convenience object is added to the Widget Palette - the Outliner, which is an XmContainer pre-configured for tree (XmOUTLINE) layout.

4.4 Widget Transformations

Designs are not static by nature. New features are added as products progress through their normal life cycles. This will mostly take the case of new dialogs, and additional options in menus and so forth, but occasionally a new feature means that an existing functionality implemented through some specific Widget class or classes needs to be upgraded to a new paradigm.

In addition, in preparing for certain cross-platform environments, a given implementation may not map particularly well. This is particularly true when considering the Motif Geometry Managers - the Form being the most awkward case; other toolkits simply don’t have anything like the Form. To effect a map that will behave in both of the toolkits under consideration, the Form often has to be replaced when preparing the translation process.

These tasks can be achieved through cut and paste - remove the old components, add and configure the new - but this is not entirely convenient, especially as it is often the case that many of the settings of the old paradigm are still useful in the new, and these have to be re-constructed.

For this reason, Widget Transformations are introduced: selected widgets can be replaced *in situ*, keeping as much of the old settings as is possible in the mapping.

Hence, we can transform a PushButton into a ToggleButton, keeping all of the old appearance, callbacks where appropriate, and label settings, at a stroke. Or turn a Form into a GridBag, appropriate to a subsequent Java mapping.

Widget Transformations are available from the Widget menu. Selecting the "Transform..." option displays a selection list of suitable transformation options for the current selected Widget context.

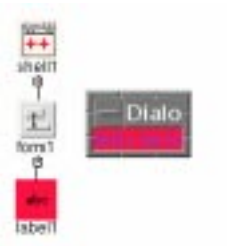
For example,

1. **Create a design containing a new Shell.**
2. **Add a Form to the shell.**
3. **Add a Label to the Form.**
4. **Set the Label String resource of the Label to "Hello World".**
5. **Set the Foreground to red.**

   Core Resources Panel, Display page.
6. **Set the Background color to blue.**

   Core Resources Panel, Display page.

   The Dynamic Display and Design Hierarchy should appear like this:[1]



1. The Foreground and Background appear inverted in the Dynamic Display, as is normal for the selected Widget.

7. **Select “Transform...” from the X-Designer Widget menu.**

   The following dialog should appear, with the list at the top containing supported transformations for the Label Widget class:

FIGURE 4-9 Widget Transformations - The Transformations Dialog

8. **Select “XmPushButton” from the list.**

9. **Press “Apply” in the Transformations dialog.**

   The Label should be replaced with a PushButton, with the resources still set, and the new PushButton still selected in the hierarchy

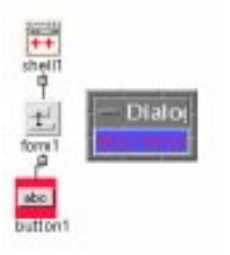


FIGURE 4-10 Widget Transformations - Post Transform

Note that not all resources can be mapped, particularly when transforming a Constraint Manager: layout may need to be effected for the new Manager widget after the transformation.

Suggested Transformations are provided on a Widget Class basis: what the Transform dialog proposes will depend on the current selection. Classes that are related through the Motif and Xt Widget Class hierarchy in some manner form the basis of the list. Thus if the current selection is a Manager widget, the list will contain alternative known Managers supported by X-Designer.[1]

1. Transforming into the Motif Composites - the Selection Prompt / MessageBox and so forth, proved highly problematic and unstable, and are therefore disabled in the current release of X-Designer pending resolution.

4.5 Reset from Top

Widget resources, as defined by the author, have an access control. That is, resources may be create-only, read-only, or read-write. For create-only resources - resources which only take effect when the Widget is initially instantiated - it may appear that the resource has no effect when applied within the relevant resource panel, and for this reason X-Designer has the Reset option available from the Widget menu - the Widget is re-created using the current state of the resource set, as though applied from the start.

But from experience, there are certain types of resource that, in addition to being create-only, have effect beyond the immediately selected Widget context. In particular, certain geometry manager Widgets have resources or constraints that cause a geometry management chain up through the widget hierarchy, as the manager requests more or less space, depending on the settings. For this reason, Reset of the current Widget is not always sufficient: sometimes you have to reset from the top of the hierarchy to allow the entire geometry management chain to take effect.

This is not always convenient, given a large design, to work in a particular context deep in the current hierarchy, then have to reset from the shell and revert back to the current context thereafter.

For this reason, the Reset From Top option is added to the Widget menu, which has precisely this effect: it reconstructs the entire hierarchy, and remembers and restores the current context.

CHAPTER 5

The X11R6 Hook Object

The Xt Hook Object was introduced into X11R6 to support runtime monitoring, debugging, and interception of Xt internal events.

X-Designer 8 allows you to use the X11R6 Hook Object directly in your applications. The following Hook callbacks can be registered within a design, to utilise the Hook functionality:

- Create

  Called whenever a Widget is instantiated by the application.

- Change

  Called when Xt invokes the SetValues method of a Widget, when the internal state of a Widget changes, when callback lists are modified, and also when the keyboard focus is moved.

- Configure

  Called when the Xt Intrinsics moves, resizes, or configures a Widget.

- Geometry

  Called when the Widget makes internal geometry management negotiation requests.

- Destroy

  Called whenever a Widget is destroyed.

While it is true that Destroy callbacks in particular can be applied to any individual Xt based widget, the Hook object makes it possible to centralise monitoring of Widget usage.

Hook Callbacks are added by selecting “Hook Object...” from within the Module menu of the main X-Designer menu bar.

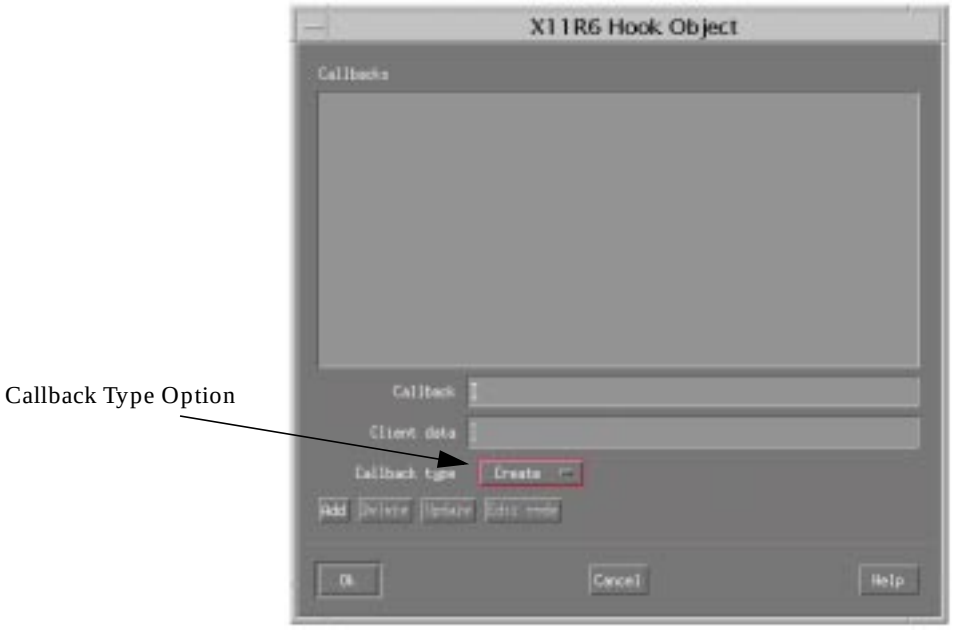


FIGURE 5-1 X11R6 Hook Object Callback Dialog

To add a Hook Callback, type the name of your callback procedure into the “Callback” field, together with any optional client data placed into the “Client data” field, chose the callback type from the “Callback type” option menu, and then press “Add”. Press “Ok” when you have added all your Hook object callbacks.

Hook callbacks are generated in a consistent way with all other callbacks in your design: a prototype is added into the stubs file, which can be edited inline or externally to X-Designer, any changes which you make being preserved across calls to the code generator.

A typical use of the X11R6 Hook object might be to add Section 508 Accessibility into your application: by catching focus change it is entirely possible to enhance the color, font, or image associated with the new focused object, or to display accessible role data associated with the focussed object, in a centralised and consistent manner. Indeed, it is possible to use the Hook object as the means of sending messages to any external agent - requesting audio feed when the focus is in a given context, for example.

For a full description of the X11R6 Hook Object, you are referred to the standard X11R6 documentation, in particular,

Programmer's Supplement for Release 6 of the X Window System,
Edited by Adrian Nye and David Flanaghan,
O'Reilly and Associates,
ISBN-10: 1565920899
ISBN-13: 978-1565920897

CHAPTER 6

Cross Platform Mappings

6.1 Java Mapping of New Widgets

X-Designer 8 allows you to generate Java code for your Motif design. Table 1 shows how each of the new widgets maps to a Java component. Table 2 shows how the new widgets map to Java Swing code.

TABLE 1 Java Mappings for New Motif Widgets

New Widget	Java Component
Spring	Panel
Box	Panel

TABLE 2 Java Swing Mappings for New Motif 2 Widgets

New Widget	Generated Widget for Motif 2.1
Spring	JPanel, SpringLayout helper
Box	JPanel, BoxLayout helper

6.2 Microsoft Windows Mappings for New Widgets

X-Designer 8 gives you the ability to create cross-platform designs to run on both Motif and Microsoft Windows, using the new Motif widgets.

Table 3 shows how the new widgets are mapped to MFC classes.

TABLE 3 MFC Mappings for New Motif 2 Widgets

New Widget	**Generated Widget for Motif 2.1**
Spring	Maps to a CWnd if structured as a class. If not, it is ignored and the child widgets are dealt with as stand-alone components
Box	Maps to a CWnd if structured as a class. If not, it is ignored and the child widgets are dealt with as stand-alone components